

# H446 OCR Programming Project

*Eshan Fadi*

*Upton Court Grammar School*

*Centre Number: 51427*

*Candidate Number: 8052*

## Abrupt Animus

## Table of Contents

Project Ideas .....	12
Engine and Language Evaluations .....	12
Unity – C# .....	13
Unreal Engine – C++ .....	13
Godot Engine – GDScript .....	13
Python (using Pygame) .....	14
RPG in a Box – Bauxite .....	14
Chosen Idea and Justification.....	15
Analysis .....	16
Problem Identification .....	16
Computational Methods .....	16
Stakeholders .....	19
SillyAustralian ® - Henry Masters (Founder and CEO) .....	19
DrunkDriving ™ - Amarveer Flora (CEO) .....	20
Research.....	21
Titanfall 2 (AKA TF 2).....	21
Half Life (and other Source Engine games).....	23
Omori .....	25
Conclusion .....	26
Interview.....	27
Essential Features .....	31
Movement .....	31
Walking/Strafing .....	31
Sprinting .....	31
Jumping .....	31
Crouching (both toggle and hold).....	31
Sliding.....	31
Wallrunning.....	32
Grappling .....	32
Double Jump .....	32
Boosting.....	32

Combat .....	32
Fire .....	32
Zoom .....	32
Swap weapon.....	33
Quick melee.....	33
Pick up weapon .....	33
Reload .....	33
Sound Effects .....	34
Saving.....	34
Scoring System .....	34
Dialogue System .....	35
Death Screens .....	35
Graphical User Interface (GUI) .....	35
Limitations .....	35
Solution Requirements .....	36
Hardware and Software Requirements .....	37
Success Criteria .....	38
Design .....	41
Decomposition.....	41
Combat .....	41
Weaponry.....	41
Swapping.....	41
Guns .....	41
Melee .....	41
Grenades.....	41
Player.....	41
Movement .....	41
Horizontal .....	41
Vertical .....	41
Wallrunning.....	42
Entities.....	42
Drones .....	42

Soldiers .....	42
Tanks .....	42
Civilians .....	42
Environment .....	43
Ground.....	43
Walls.....	43
Pickups .....	43
Interacts.....	43
UI .....	43
HUD.....	43
Menu .....	43
Title screen .....	43
Options .....	43
Pause .....	44
Save menu.....	44
Checkpoints .....	44
Load game .....	44
Dialogue .....	44
Interpreter.....	44
Actors .....	44
Managers .....	44
Structure Definition .....	45
Algorithms.....	45
Movement .....	45
Enemy Entities .....	51
Weapon and its Interaction.....	54
Dialogue .....	57
Save System .....	62
Usability.....	64
Game Design .....	64
First Designs:.....	65
Main Menu:.....	65

Save Slot Menu:	66
Game HUD:	66
Choice:	69
Subtitles:	69
Pause Menu:	70
2nd:	70
Main Menu:	70
Save Slot Menu:	71
Pause Menu:	72
3rd:	72
Main Menu:	72
Save Slot Menu:	73
Final Agreed Designs:	73
Main Menu:	74
Save Slot Menu:	74
Game HUD:	75
Choice:	75
Subtitles:	76
Pause Menu:	76
Variables and Data Structures	76
Enumerators	81
PlayerMovementState:	81
DerivativeType:	82
MemberState:	82
Databases	82
Dialogue:	82
SceneInfo:	82
Naming Conventions	83
Class Diagrams	84
Entities	84
Weapons	85
Iterative Tests	85

Strafing.....	86
Looking.....	87
Boosting .....	87
Jumping .....	88
Crouching.....	89
Different crouch keys .....	90
Looking and strafing combined tests .....	90
Fire.....	91
Reload.....	92
Swap weapon .....	93
Add weapon.....	93
Weapon pointing to target.....	94
Spike Testing.....	95
Entity Movement .....	96
Entity Searching .....	97
Entity Combat.....	98
Saving and Loading .....	99
Save Data Application .....	100
Displaying Saves Normally .....	100
Displaying Saves when making a new save slot.....	101
Selecting a new menu .....	101
Player Death .....	102
Quitting .....	102
Loading Screen .....	103
Moveability and Cursor state when loading a scene .....	103
UI Audio.....	104
Volume Slider .....	105
FOV Sliders.....	106
Wallrunning .....	107
Transpiling a line .....	108
Dialogue Interaction.....	109
Post Tests.....	110

Robustness .....	110
Glitches.....	111
Strafing.....	111
Jumping .....	112
Crouching.....	113
Using weapons .....	114
Taking damage .....	116
Saving.....	117
Dialogue .....	117
Entity behaviour .....	119
Usability Tests.....	120
Development .....	121
Prototype 1 .....	121
Input Detection .....	121
Movement .....	122
Interview .....	130
Interview .....	135
Miscellaneous .....	151
End of prototype tests .....	152
Strafing .....	152
Looking .....	153
Boosting.....	154
Jumping .....	154
Crouching .....	155
Different crouch keys .....	155
Looking and strafing combined tests .....	156
Interview.....	157
Prototype 2.....	157
Interacting with weapons.....	157
Entities (Damage).....	167
Interview .....	173
End of prototype tests .....	185

Entity Movement.....	185
Entity Searching.....	187
Entity Combat .....	187
Interview.....	188
Prototype 3.....	189
Automatically Reloading.....	189
Held Zoom.....	190
Weapon point to target .....	191
Saving and Loading System .....	194
Interview .....	209
Menus and level transitions .....	210
Miscellaneous .....	224
Referencing the Game Manager.....	224
Referencing the camera .....	226
End of prototype tests .....	226
Saving and Loading.....	226
Save Data Application.....	227
Displaying Saves Normally .....	227
Displaying Saves when making a new save slot .....	228
Moveability and Cursor state when loading a scene.....	228
Interview.....	229
Prototype 4.....	229
Button to loading screen.....	229
Interview .....	231
Audio.....	232
Options .....	236
Interview .....	238
Movement .....	240
Interview .....	240
End of prototype tests .....	251
UI Sounds .....	251
Volume Slider .....	252

FOV Slider .....	253
Sliding.....	254
Boosting.....	255
Double Jumping.....	255
Wallrunning.....	257
Interview.....	258
Prototype 5.....	259
Dialogue .....	259
Interview .....	274
End of prototype tests .....	277
Transpiling a line .....	277
Dialogue Interaction .....	277
Interview.....	279
Evaluation.....	279
Evaluative Testing .....	279
Robustness .....	279
Evidence – Normal.....	280
Evidence – Intense.....	280
Glitches.....	281
Jittery movement .....	281
Weapon duplication .....	283
HUD Disappearance .....	284
Dialogue if skip .....	284
Strafing.....	285
Jumping .....	290
Crouching.....	293
Boosting .....	295
Wallrunning .....	297
Using weapons .....	300
Taking damage .....	304
Saving.....	306
Dialogue .....	310

Entity behaviour .....	312
Usability Tests.....	314
Evidence – There is no game.....	316
Evidence – Opening the game, Main Menu .....	317
Evidence – Option Menu in the Main Menu .....	317
Evidence – Empty Slot Menu.....	318
Evidence – Filled Slot Menu.....	318
Evidence – Loading Screen.....	319
Evidence – HUD .....	319
Evidence – Pause Menu.....	320
Evidence – Options menu in the Pause Menu .....	320
Success Criteria .....	321
Interview.....	322
Limitations and Maintenance .....	322
Bibliography.....	326
Appendix (All the game's code) .....	328
UISounds.cs .....	328
CASCapsuleVisualiser.cs .....	329
FPS.cs.....	329
IsGroundedTest.cs.....	330
PosAndVelocity.cs .....	331
DialogueInteraction.cs.....	331
TreeNode.cs .....	346
Entity.cs .....	346
PlayerEntity.cs .....	348
EnemyEntity.cs .....	349
Drone.cs .....	357
Tank.cs.....	359
Soldier.cs .....	360
EnemyManager.cs .....	360
AudioManager.cs .....	360
DialogueDatabaseManager.cs.....	361

DialogueManager.cs .....	362
MenuManager.cs .....	364
PlayerManager.cs .....	367
Options.cs .....	373
PauseGame.cs .....	374
BodyDetection.cs .....	375
PlayerGunInteraction.cs .....	375
PlayerLook.cs .....	381
PlayerMovement.cs .....	382
Checkpoint.cs .....	394
LevelBullets.cs .....	395
LevelHighscores.cs .....	395
PlayerData.cs .....	396
WorldFlags.cs .....	398
SaveData.cs .....	398
SaveSystem.cs .....	399
SerializableStructs.cs .....	403
GUISlots.cs .....	404
LevelLoading.cs .....	406
NewGame.cs .....	407
SelectSlot.cs .....	407
SlotSelection.cs .....	407
GlobalConstants.cs .....	408
Interactable.cs .....	408

## Project Ideas

Idea	Short Explanation	Complexity	Language	Computational Methods
Movement first-person-shooter game	A game where the player must simultaneously manage their momentum and their aim	Medium - ample use of math and GUI. Movement must be fine-tuned along with movement-based bug fixes. Player physics, and bullet physics, enemy AI, and modular interconnected components, OOP, libraries	C#, C++, GDScript, Python, easier with engines	OOP, database, modular design, abstraction for the GUI, reusability for functions
Game engine	An application designed to help and aid someone in the creation of game or another app	Difficult - lots of math, OOP, GUI, implementing a physics engine, along with functionality for any component, must include compilers and built in libraries to assist in building the game	C#, C++	OOP, decomposition of functions
Programming Language	Writing a language and run it either being compiled, or to interpreted	Easy - filled with trees and if statements, and exception raising, must have built in functions, handle memory and add basic data structures	C#, C++, CPython	Selection, trees, decomposition, and modular design for various parts of the compiler

## Engine and Language Evaluations

The purpose of an engine within software development is to speed up the development speed. Usually, these contain tools that manage and automate significant background processes, such as the script execution order, a built-in physics engine, and a lighting pipeline. Many modern engines feature a modular design for their workflow, with there

being objects which you can attach components to, to add further functionality to the object. Some of these may be colliders, materials, or even scripts.

## Unity – C#

Released in 2005, Unity is an extremely useful engine that has great capabilities for both 3D and 2D, with powerful tools to assist in the creation of apps (mostly games) and experiences. It has a huge community surrounding it, making it incredibly easy to learn, with a wide variety of plugins to aid in the development of certain aspects of the process. The currently supported language used within it is C#, part of the C family. The language is quite easy to learn, having aspects like Python and JavaScript. It also simultaneously has the speed and access of C++, as it is a compiled language. I have some previous experience in using both Unity and C#, but the majority of their functionality is still to be explored. However, Unity recently made some economic decisions that have severely damaged the reputation of the company and software. As such, if I wish to continue the project afterwards, I may end up having to completely transfer the entire project to another piece of software, so this may not be the best long-term solution.

## Unreal Engine – C++

All the way from 1998, this engine is well known for boasting incredible visuals without much effort required from the developer. It is comparatively easier to optimise the application to run faster, as certain mesh optimizations exist natively to Unreal Engine 5 (namely Nanite). Proven its ability repeatedly, the Unity scandal has brought more eyes upon this amazing piece of software. However, due to its fabulous visual capabilities by default, making lightweight applications and games is far tougher, as many layers must be stripped away. Furthermore, it is well known that Unreal Engine is not built for the 2D experience, limiting the type of content that it can create. The language of choice for UE5 is C++, a language notorious for being extremely difficult to learn with low-level concepts such as pointers mixed in with high-level abstractions such as object-oriented programming. I have little-to-none prior experience programming with C++, so the learning curve to use UE5 is far greater than the other options.

## Godot Engine – GDScript

Godot is the game engine equivalent to Blender 3D in the 3D modelling world - a completely free and open-source alternative, which is only getting better and better every update, and will most likely become the industry standard within a decade. It uses a node-based system to construct the world and all objects which is similar to the systems that other game engines use. This intuitive system allows for more flexibility,

greatly helping in the learning process. Godot mainly uses GDScript, a custom language which the engine supports completely for all platforms. It also supports C# and C++, but C# is not currently available to be compiled for the web in Godot 4.2 (the latest version of Godot at the time of writing). Godot can also support any other language using extensions to the engine. The worthwhile option would be GDScript, which is similar in syntax to JavaScript and Python, and has many built in functions such as `is_on_floor()` to handle common and simple checks automatically. The number of tutorials that the community and the Godot team themselves have made is vast and is only increasing, as many people flock from Unity to Godot, making it so that learning the engine and its language would be far easier than before.

## Python (using Pygame)

Python and Pygame together allow anyone to easily pick up and create anything and everything that they want. Unlike the other entries on this list, this is not an engine, which means that the creator must program in every feature, from collision handling and response, to manually testing individual locations for objects, instead of placing them in a visual scene and moving them easily. This significantly slows down the development of the application and can be extremely tedious as everything must be done by the developer, reducing the amount of creativity and passion that can be put into the project. While the language itself is extremely easy to learn, the underlying concepts behind game development such as matrix transformations and quaternions are extremely difficult to understand and waste one's time when implementing them in such a small timescale. Such a project would be better suited to one with a far longer time frame to work within.

## RPG in a Box – Bauxite

Made using Godot, RPG in a Box is a game engine that has everything built within it for you: a voxel editor, fully working dialogue system, a sound effects generator, and even a visual scripting language. While many of the other engines on this list also have visual scripting as part of their arsenal, none use it or recommend it as much as RPG in a Box. It also has a secondary fully working scripting language called Bauxite, which is extremely similar to Lua, a famous programming language that has been used to make amazing games such as Hades. The engine is designed to be as friendly to a normal person as possible, such that no prior experience is required to even begin making something within this engine, but this also happens to be its downfall. Since so many features are required to use the built-in system to have full functionality, it becomes harder to implement higher level concepts without effectively writing your own engine.

## Chosen Idea and Justification

The game engine is far too complex and time consuming for the constraint of time, while also requiring more learning than either of the other options. The programming language on the other hand, is simpler and would not allow me to easily incorporate as many computational methods. The movement first-person-shooter game is a great medium between the complexity of the two other options, while also allowing me to incorporate a great deal of creativity and interesting implementations of the course content.

Furthermore, I will be using Unity to implement this since I have prior experience using the engine and C# as well, so I can spend more time focusing on constructing the project than learning how to use the program itself. I also get the benefit of using an engine which allows me to focus on unique ideas, rather than implementing standard algorithms and techniques.

# Analysis

## Problem Identification

An up-and-coming group of the gaming community is a collection of speedrunners, craving for fast movement and fluidity to beat the games faster and faster. To satiate this need, I have decided to construct a fast-paced movement first-person-shooter game, demanding both an understanding of momentum and precise aim, to hit a selection of enemies and reach the end of the level as fast as possible. The game will also have a secondary genre of a role-playing game, allowing players to progress and unlock more movement abilities and upgrade already existing ones to navigate the game at faster rates. Applying restrictions to the upgrades players can apply at one time will force them to strategically use their available abilities wisely and changing their style of play based upon the current scenario. For example, puzzle-based sections may better be played using a set of upgrades (typically called a build) that benefits movement and its control. On the other hand, while in combat it could be better to use a build that increases one's health and damage dealt.

## Computational Methods

Method	Why	How
Thinking abstractly	Referring to the process of separating ideas from reality, it can be understood in two separate ways: abstraction for the developer, and abstraction for the player. For the developer, abstraction refers to the difference between the code and the hardware and hence creating a model that will make the best use of the hardware. For the player, abstraction is the difference between the information given to them (such as the GUI, colours of effects) and the actual code and logic behind it all.	Only the bare minimum information is provided to the player, making sure that they only know what the developer wants them to know. Workings of certain mechanics could be summarised within a paragraph or shown through cutscene tutorials. Information on weapon states can be shown with images instead of text. As much as possible should follow the show-not-tell principle and detail does not need to reach that of realism.
Thinking procedurally	Otherwise known as decomposition, this is the process of breaking a large problem into sub-problems, making the overall task much simpler. Without the ability to	The task of creating a game can be split into the many parts of game development, such as programming, art design, and other things. Within programming, it can be used to split the game into

	<p>decompose, designing and creating a game would be completely impossible. People already decompose problems to solve them. By splitting a huge problem into extremely simple ones, it makes the entire task far easier and reduces the time wasted in trying to solve a complicated problem. All the saved time will create a better product in the end.</p>	<p>separate modules, classes, and functions. Grouping together related functions with importable modules will reduce the amount of code one needs to write and is better to maintain and debug. It is also far easier to create a function that does only one job and have a separate function call all the pieces when they are required, avoiding code duplication and errors that might occur when doing so.</p>
Thinking ahead	<p>When designing programs, rewriting the same code or similar code over and over again can be boring, inefficient, and a pain to debug, while also reducing the readability and increasing how error-prone it becomes. It also becomes far faster to program if all the steps have been planned out beforehand, such as inputs and outputs. Identifying spots where one can cache data to improve data access speeds and reducing memory usage will also allow the game to run faster, while leaving more resources for other parts of the program. Thinking ahead also references planning out the timing of when to implement features and how long it should take to do so.</p>	<p>One can simplify repeatable code by extracting them into functions and their respective calls. Taking this a level up means using classes (and hence Object-Oriented Programming) to reduce code duplication as much as possible. This fits in with using Unity and C# since it uses objects as the framework, and so programming using OOP will be far easier and better supported. When multiple objects may access the same data, this data will be cached in a static section of the class, increasing memory efficiency. Techniques like polymorphism and abstract classes will further reduce the need to rewrite code.</p>
Performance modelling	<p>In games where skilfully timing and predicting trajectories is the aim, having any source of freezing or lag (when there is a significant difference between the player's inputs and the outputs displayed on the screen) will result in a subpar, even rage-inducing experience, due to issues out of the player's control. While this is the intention for some games, it is not the intention for mine, and as such, code should be written efficiently to improve the player's</p>	<p>While doing the programming can be more enjoyable than other parts of the development process, wasting time testing inefficient and longwinded solutions can be a massive bane that may end up costing you the project. Being able to evaluate the performance of a function or script before it is rigorously tested by modelling it and quantitative analysis will help identify weak points in algorithms and potential places to improve the efficiency, reducing the lag a player</p>

	<p>experience. Performance modelling will help in analysing inefficiencies and hence places that could be rewritten, resulting in an overall all better game.</p>	<p>might experience. One useful tool to quickly measure a function's efficiency is Big O Notation, showing how effective the algorithm is as the input size increases.</p>
Thinking concurrently	<p>Not every process in a game can run sequentially. Sometimes things must run simultaneously to run properly, such as physics calculations alongside GUI rendering, and especially input sensing, which must be handled concurrently with other inputs, such as moving and firing. Also, waiting for other tasks to finish or acquire data can block the CPU, which is more wasted time, so allowing the CPU to switch to another task while it waits is efficient use of its cycles.</p>	<p>The Unity Engine already manages a multitude of processes concurrently by assigning them to separate threads, which can then be picked up by separate cores. Other tasks such as highly parallelisable numerical tasks and graphical tasks can be offloaded to the GPU, which performs the same code on many different data (SIMD architecture) in a huge array of parallel cores.</p>
Thinking logically	<p>Without the ability for the program to change what it performs based upon the inputs, many projects and pieces of software would be virtually impossible to create. The ability to loop code also significantly reduces repetition of code, reducing the error rates from poorly duplicated code, and making it easier to debug.</p>	<p>Making different functions run when different input keys are pressed is the entire backbone of games, and as such, having the ability to change control flow is incredibly important. Not only that, but having code run on certain conditions being fulfilled makes sure that only necessary code is run and avoids potential crashes. For example, if a raycast (shooting an imaginary ray and checking if it hit any object) hits an object, the pickup animation might be different for different weapons if the object hit was even a weapon in the first place. In certain situations, however, using a switch statement is more efficient than using many if statements, since in a compiled language, the compiler simplifies switch statements into a switch table, which is a table of all the relevant memory locations for the different parts of code, and a hashing function is used to map the</p>

		match to the memory location. This ability is more efficient than evaluating many if statements. Just as integral is the ability to have all the game code, or at least the relevant sections, be rerun at the start of the next frame, allowing for an interactive experience and to make sure that the game can actually be played. Looping can be used elsewhere within movement calculations and predictions or for iterating through lists and arrays.
Backtracking	When programming, it is not always sure beforehand whether a solution will work, or if it is worth implementing a certain way. As such, you may have to backtrack to previous working solutions and try a different approach to settle upon a far better solution. Over time, this maintains or improves the code, will also keeping the development enjoyable, and motivation high.	Sometimes, potential optimisations may be more hassle than the old solution and one's time could be better spent elsewhere. If at the time such a solution does not present itself, you can leave it for the time being and come back to it at a later date, when your brain is refreshed and other parts of the code may be completed. This allows you to come up with fixes that you may not have thought of before, since you can leverage the other code, and a fresh mindset.

## Stakeholders

A stakeholder is a person/organisation that is interested in a project, in this case, my game. These people have the power to change significant parts of the documents as they are the ones who the developers must consult when implementing certain features and how those features will work. For example, these people might be users of the program, such as the players of a game, or they could be producers and publishers wishing to make money from selling the program.

### SillyAustralian ® - Henry Masters (Founder and CEO)

My main stakeholder is the founder and CEO of the amazing game development company, SillyAustralian ®, Henry Masters. Masters is a huge fan of precision platformers with lots of challenge, as they prioritise a skilful experience that one can get better at, constantly keeping them within the flow state. Some examples are Celeste

and Super Meat Boy, both being loved by players of the precision platformer genre. However, they also like role-playing games (RPGs), such as the hit 2015 indie game, Undertale from the brilliant mind of Toby Fox, and The Legend of Zelda: Breath of the Wild from well-known console and game production company, Nintendo Co., Ltd.

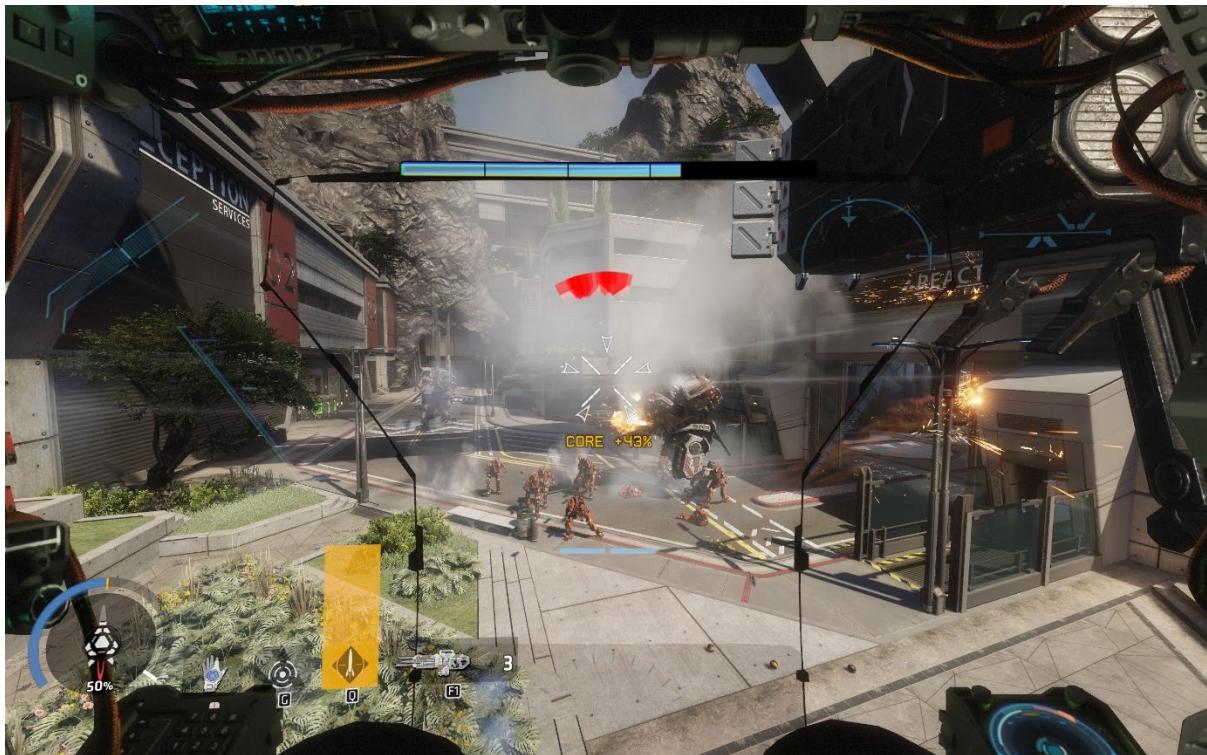
Recently, the company is wishing to start adding a secondary part to their business by publishing other small games under big names and help with the marketing of the experience to allow the developer to reach a larger audience than they may have gotten without going through the publisher. As such, they are willing to provide feedback on the actual code of the game, making sure it is optimised for various devices which may or may not have the greatest hardware, preferring efficient solutions as well. This leads to them requiring a game that runs fast and smoothly as their top priority, since the game should be accessible to as many people as possible, and they are willing to have compromises on the visual clarity of the game and the background story of the game as this does not contribute to the gameplay experience. They would also very much all the code to be easily maintainable, as they wish to continue developing the game after with their own software team. This means that I have to use many easy-to-read notations and naming styles.

### DrunkDriving™ - Amarveer Flora (CEO)

My secondary stakeholder, handling the playtesting of the game and complexity of the playstyle is Amarveer Flora, the CEO of DrunkDriving™, along with a small selection of his employees. Also a fan of difficult combat in games such as Elden Ring and Dark Souls, they have experience with massively multiplayer online role-playing games (MMORPG) such as Hypixel Skyblock and Black Desert Online. Similarly to these games, they also like the simple but logical gameplay that comes with automation games like Beltmatic and the Minecraft modpack, Techopolis 2. Importantly, they have great experience with shooter games like Overwatch 2, Fortnite, and Call of Duty, while having great reaction times. This makes them a great stakeholder to test the game for bugs while being able to play at the recommended difficulty of the game. They similarly prioritise the player's gameplay experience, but they are more focused on the bugs and the instability of the game. Having many bugs in common gameplay will lead to an unfair and dissatisfactory play, so their priority is fixing as many of the bugs as possible, such that a player does not ever experience a frustrating playthrough. They are also willing to get rid of certain features such that the core gameplay elements are kept, and so they will be secondarily overseeing the features added to make sure priority is given to the features the player will interact the most.

## Research

### Titanfall 2 (AKA TF|2)



(Fighting as a titan in Effect and Cause)

Released on 28<sup>th</sup> October 2016 by Respawn Entertainment and published by Electronic Arts, Titanfall 2 is movement shooter, where the player swaps between moving around as a human character called a pilot and playing within a Titan, where the style of play completely changes. As a human, the player has access to wallrunning, sliding, double jumping, and other abilities depending on the level. As a Titan, the player is able to gain great combat abilities, specific to the kit that they choose to apply to the Titan, all of which overpowers a human in every aspect. However, the movement capabilities of the Titan are reduced, leading to the only shared ability between all Titan kits to be the ability to move around. Some kits can fly, others can dash, but most of them have no extra movement capabilities. The 20-foot-tall war machines also struggle to get in some places, and as such the player will need to frequently exit them to reach other places or remove large volumes of health from enemy Titans via hijacking. This excellently illustrates the design philosophy for each of the levels in TF|2 – ‘211’: two parts pilot combat, one part pilot movement/puzzle solving, and one part Titan combat.



(A time-based puzzle in *The Beacon* where the player must activate the platforms correctly using this level's special feature, the Arc Tool)

When designing levels, the team at Respawn Entertainment made each of them based upon on a different feature. For example, in the level Cause and Effect, the player swaps between the past and the present, managing enemies in both timelines, and using the changes between the two to help in accessing different areas. This planning technique is something that would be wise to follow, but since levels are designed to be completed in speed (and making long levels can be difficult) it might instead be smarter to split a theme across multiple levels, which are typically called zones. It should not be overdone however, as the feeling of 'wanting more' is always better than the feeling of boredom.

## Half Life (and other Source Engine games)



(Freeman fighting against a headcrab in Office Complex)

Half Life was the inspiration for many of Titanfall 2's puzzle design, where there is a minimum amount of momentum required to traverse and solve the puzzles. Released in 1998 as Valve Corporation's first game after 2 years of work, Half Life set the standard for what an amazing story-based first-person shooter game should be. It carefully and cleverly introduced the player to all its mechanics, and showing the depth that they could be used. Across the 19 chapters, and approximately 12 hours of immersive gameplay, Valve crafted an experience that tells the story of the creation of a dystopia, and Gordan Freeman (a young scientist) along with his attempts to stop it.

Valve built their debut game on GoldSrc – an extremely modified version of the Quake Engine and the Quake II Engine from id Software (the developers estimate that the resulting code is over 75% their own). The Quake Engine had a feature within its movement calculation code called 'air acceleration', which as its name suggests, allows the player to accelerate in midair. Normally, this wouldn't matter, since the drag from moving forward and the deceleration that you experience when touching the floor is enough to cancel all speed increase. However, with a special movement technique called air strafing and bunnyhopping, it's possible to gain speed, essentially to as fast as you want. Valve later modified and upgraded this to make the Source Engine which debuted alongside Half Life 2 and Counter-Strike: Source, and later on the Source 2 Engine with Half Life: Alyx. Since all the later engines were based of the previous engines, they all contain the same similar movement code and hence speedrunning

techniques, which is incredibly useful for speedrunners since they can reuse their own skills.

The Half Life franchise is also well known for its incredibly realistic AI, which is especially well demonstrated with the Overwatch Soldiers from Half Life 2 and the following games in the series. The soldiers are capable of replicating and enforcing blitzkrieg-style military tactics against the player and making smart decisions in the usage of their weapons. There is also a clear hierarchy present within each team, and the behaviours of the different soldiers are cleverly explained behind the story of the game.



*(The Combine AI has decided to seek cover with its teammate, and reload under cover)*

The Combine AI functions as a multi-part system, with each enemy following a set of logical steps, then reporting back to a manager system, which assigns the respective enemies the appropriate roles based on the number of targets. Each soldier will then maintain that slot until their schedule finishes, which is a set of tasks assigned to it, such as finding cover, or firing at the player. This complex system allows for the tactics that the enemies should enact, while also keeping the situation manageable for the player, since the system will not allow more than 2 enemies to attack the same target simultaneously, which makes it a brilliant basis to design the enemy AI for my game as well.

## Omori



*(Mari talking to her brother, Omori, who is also the player)*

An absolutely phenomenal role-playing game (RPG) by OMOCAT using RPG Maker MV, Omori tells two interconnected stories of two connected teenage boys, dealing with both friendship and guilt. The bulk of the game is spent interacting with characters in both the storylines and completing quests for all of them. Dialogue systems in games can be incredibly complex and important, giving one an insight into how the characters of the game is feeling, and the possible options can change based on what the player has done. If the player has completed a quest, the system must be able to detect the change to the relative variables and change accordingly.

The importance of dialogue in a game comes not from the words, but the feelings they invoke and pass on to the player. After all, if the only things a character says is just describing the current situation, better world design or even a signpost is what you need to have ‘dialogue’, instead of implementing such a complicated system. However, the inverse is not true – a signpost cannot represent emotion as well as a set of characters, no matter whether they display the emotion or not.



*(Kel takes damage from a forest bunny, one of the enemies the player is currently fighting)*

Omori's system links together facial emotions and text effects to further accentuate the feelings of the characters. I, on the other hand, do not need to invoke emotions through facial expressions, nor do I need such extravagant text effects such as shaking and waving. However, applying colour, italicising and bolding text is useful in pointing out the most important things to the player while they are skimming through the text, while also helping them retain those specific pieces of information. In the game that I am making, the player will be under a lot of stress, so being able to skim dialogue quickly is important.

## Conclusion

From my research into these different games, I have decided on different elements of these games that would fit well into this game. From Titanfall 2, the main inspiration for this game, the way it splits the game into long chapters will be utilised. The ability to run along walls will also be used. However, the majority of the movement will be taken from Half Life and the various other source games since this is the basis for even TF2's own movement. I will also be utilizing part of the design philosophy (211), splitting part of the levels into both combat and the parkour section where the player has to get around to different locations.

From Half Life 2, I will be utilising part of the enemy's AI system, although it will not be implemented to that extent. Rather, the manager AI will only exist to alert other enemies' part of the same group of the player's existence. This is necessary to keep the project within a reasonable timeframe.

Omori will be the inspiration for the dialogue system, although only text will be on it: no images or animated faces since the dialogue is only a secondary aspect of the game, not meant to detract from the main shooter aspect of the game, since the RPG aspect is just a sub-genre.

## Interview

For this interview, I needed to gain an insight into the exact features that the stakeholders wanted, and as such I convened with both. First, I prepared some questions, spread across different categories:

- Movement
  - o What should be the basic movement options?
  - o Any special movement options?
  - o Should movement be location specific?
- Combat
  - o What should the basic combat abilities be?
  - o Should the player have multiple weapons, and if so, how many?
- Audio
  - o Should there be sound effects?
  - o Should there be a soundtrack?
- Systems
  - o Dialogue
    - What should the system be capable of?
    - Should there be quests?
  - o Saving
    - Is this system needed?
    - Should it be slot-based?
  - o Currency
    - Will the game have any currency?
    - What should the currency be used for?
    - Is this a necessary system?
  - o Any extra systems?
- World
  - o Should the game be split into levels?
  - o Should enemies be placed in predetermined locations in the world?

- Should the world be modelled, or just prototype models?
- UI
  - Should there be a main menu?
  - Should there be a pause menu?
  - Should there be a HUD?

Upon asking these questions, I was able to obtain the following responses from both my stakeholders:

**What should be the basic movement options?**

The player should be able to move both slowly and quickly in all directions on the floor. They should also be able to jump off the floor. They should also be able to crouch, both as a hold key and a toggle, where the held version should take priority.

**Any special movement options?**

Special movement options should be the ability to run along walls, an ability to grapple to any surface in the world, an ability to slide when the player is running fast enough. A slide should provide a speed boost for the first second of entering it. Being able to jump in mid-air and multiplicatively increase their speed is also useful.

**Should movement be location specific?**

No, the movement should be useable anywhere, with only the ability to run along walls being specific to walls only and not the ceiling.

**What should the basic combat abilities be?**

The player should be able to fire bullets from a gun, and be able to reload the gun, if infinite ammunition is not an option. Zooming in would also be useful so the player can have higher precision when firing. The player needs to also be able to swap their current weapons with the ones on the ground or dropped from enemies if possible. Finally, an optional add-on is the ability to quickly perform a melee upon any enemy using the current weapon in hand.

**Should the player have multiple weapons, and if so, how many?**

The player should have access to hold onto multiple weapons but should only be able to use 1 at any given moment. A nice middle ground between too many and too few is 4 weapons. This should work well with the fact that weapons should be of 3 types: blades, explosives, and bullets.

**Should there be sound effects?**

Yes, the game needs to have sound effects to indicate if an action has taken place. However, you don't need to design the effects, just put in the systems to add them in place.

Should there be a soundtrack?

No, the game does not need to have a soundtrack, it should just use the same system as the sound effects, but you need not implement it.

What should the dialogue system be capable of?

It should be able to display text on the screen at a minimum and have easy syntax so that we can have people writing the dialogue with no other knowledge of how the rest of the game works. Optionally, it can include things like choices for the player and animation keys for characters to face other characters.

Should there be quests?

No, a quest system is not needed and is also optional in combination with the dialogue system's choices.

Is the saving system needed?

Saving is integral for any player in any game and should be one of the higher priority systems that would be implemented.

Should it be slot-based?

Many games do use a slot-based system where you can only have a limited number of saves at a time. We are not a fan of this system however and would like the player to be able to make an infinite number of slots, with its own data.

Will the game have any currency?

The game should have the capability for currency, granted to the player upon killing enemies, and performing other important actions.

What should the currency be used for?

The currency should be used to buy upgrades to the player, such as more health, and more health regeneration.

Is this a necessary system?

No, it need not be implemented at all and is completely optional. This system should be implemented last

Any extra systems?

It would be interesting to have a system to challenge the player into getting through a level as fast as possible or perhaps deciding to survive as a pacifist. This could be a scoring system. It would be interesting to make the game dynamically make itself harder as the player plays as a pacifist more and more: this could tie into some story behind the game, such as calling it a glitch meter and making the player a robot.

Should the game be split into levels?

The game should be split into multiple different levels and have the capability to set up more levels as much as possible.

Should enemies be placed in predetermined locations in the world?

This is both yes and no. For replayability, the enemies should have partly random spawn locations, but these should all be within predetermined locations, so that speedrunners can enjoy speedrunning the game more, while still having a level of difficulty to the run.

Should the world be modelled, or just prototype models?

Leave everything as simple objects, and we shall replace them with our own models once you pass the project to us (SillyAustralian). This covers all graphical elements; they need only be placeholders.

Should there be a main menu?

Yes, the player needs somewhere to start the game and decide which save to play, along with managing settings or quitting the game. The buttons in this menu, and any other following interactable menu, should produce sound effects when hovered over and pressed on with the left click.

Should there be a pause menu?

The player should be able to pause the game, stopping all actions including the game's physics and enemy attacks. This menu should allow the player to manually save and load, if implemented, and allow them to change their settings and finally quit to the main menu or quit the game.

Should there be a Heads-Up-Display?

Absolutely! The HUD is very important for the player to see some information about data about themselves that they need to keep track of. This can include the player's health and their currently equipped weapons, along with information related to that weapon.

## Essential Features

### Movement

#### *Walking/Strafing*

Walking is the basic movement action that the player can enact and will need this to start of many other actions. Comparatively to other ground-based movement, the speed of this will be situated in the middle of the rest.

#### *Sprinting*

While walking can be all one might need, getting to places faster is important to avoid getting hit in combat, or complete certain puzzles within the required timeframe. Sprinting will increase the ground acceleration that is enacted, and the player will also have a higher top speed compared to walking.

#### *Jumping*

Another staple of the movement actions, jumping allows the player to easily gain vertical momentum, and it can be developed by allowing the player to jump while in the air, or jumping off walls.

#### *Crouching (both toggle and hold)*

Crouching will shrink the player's hitbox vertically, granting access to smaller passageways, and providing a potential advantage in combat, since the area that enemies can attack the player in is far smaller. To balance this out, crouching will force the player to move much more slowly and will be the slowest ground-based movement option.

#### *Sliding*

When above a certain speed, using the crouch button will instead lock the player into a sliding state. Sliding should first increase, then decrease the player's speed, encouraging you to leave your slide earlier to gain speed. The target direction will be the current direction of movement, but input will only allow slight nudges in direction, rather than a controlling factor, allowing the bare minimum control. Friction while in a sliding state will also be a lot lower than other states.

### *Wallrunning*

A much more advanced movement option, this makes the player stick to walls and allow them to move along it. This opens the possibilities of combat terrains and puzzles and is usually included in movement-shooter games. The player can also jump off these walls, which may lead to a variety of solutions to different situations the player may encounter.

### *Grappling*

While not as common in other games, grappling is still fun to use, but is also far harder to control, since this requires one to plan trajectories. It is also much harder to implement but is still well enjoyed by many players in the game's it shows up in, such as the multiplayer mode of Titanfall 2, or as a basic movement option of Ghostrunner.

### *Double Jump*

This is a far more common ability, where while in air, the player can press the jump button again to gain an extra boost vertically, akin to a jump on the ground. Without this, the player is only permitted to jump while touching the ground.

### *Boosting*

Like a dash that might appear in other games, a boost is a multiplicative increase to the player's speed and will allow them to gain speed much faster than attainable through other methods. Due to the potential of exponential speed, both a maximum limit for boosting must be set, while also implementing a heavy cooldown for using it.

## Combat

### *Fire*

On the combat side, firing is an essential action. Without this, it would not be called a movement-shooter. Firing will be a catch-all term for using weaponry, such as slicing with a blade, shooting a weapon, or throwing a grenade.

### *Zoom*

While a player may not use this as much when they are moving around a lot, it is still a staple of the shooter genre to include some sort of zoom capability on weaponry or on your visuals to be able to see long distance clearer. This may have a side effect of

encouraging a stealthy and cautious playstyle instead, but this could be negated by having more difficult sections later in the game.

#### *Swap weapon*

From the beginning of first-person shooters, the usefulness of having multiple weapons and the strategy that appears from doing so was always an incredible lure into the games of the past. This inherent feature stuck through the generations and remains as an expected feature in a shooter. However, starting from Halo, the number of weapons one could carry could be limited, due to the limitations of the controller at the time. As such, another trend of having a limited number of weapons that the player could equip at one time rose in popularity. It's an incredibly easy way of bringing depth into any shooter, by making the player have to choose which weapons, they want to use, and by having a limited ammunition capacity, also making the player choose if they can wait to get ammunition, or if they need to change weapons now.

#### *Quick melee*

While unlikely that the player might repeatedly get within close range of any of the enemies, there must be a close, high damage option for the player to enact. The melee is a powerful move, that depending on game-to-game, can be one of the highest damage options available. This ability allows the player to use any weapon to inflict a strike onto an enemy, no matter the usual use of the weapon.

#### *Pick up weapon*

Since the player can run out of ammunition, it is likely that they will end up without any weapons with ammunition in them, other than blades. As such, it is possible for the player to pick up weapons dropped by enemies or placed in boxes throughout the level to counteract this difficulty.

#### *Reload*

Guns do not have a limitless capacity of ammunition, and to help players to immerse better with the game, the player can reload the magazine in the currently active weapon if applicable to do so. This can add a further layer of strategy to the game, as a player can decide to reload in the middle of a magazine before entering a gunfight or not do so if they fear they will get hit in a moment.

## Sound Effects

The brain is very good at linking different senses together for the same piece of information. Having different sound effects (sfx) link to different actions will act as either a confirmation that the action has been processed, or it may warn the player of danger.

## Saving

Saving is needed for longer games, since players can get incredibly frustrated with losing progress every single time they close the game. As such, saving has been implemented in many major games as a basic feature. Different games do the system slightly differently, usually splitting into 3 different types. The first is where the player has many slots and can save the game into any of these save slots and have multiple different places they can pick up from. This can also be used to have entirely different playthroughs existing at the same time. The second method has only one save slot but saves progressively by making a new save every single time. This only allows for one playthrough, but many checkpoints that the player can load back into. The third method is the one that I shall implement, where a mix of the other two types is used. Specifically, the player will have multiple slots, where each slot denotes a playthrough. Each slot will have multiple versions based on manual saves or automatic ones based on checkpoint.

## Scoring System

Each level is scored based on:

- Time to complete.
- Enemies killed.
- Secrets collected.
- Inverse of death count.

Further point bonuses will be granted for:

- Using less weapons (melee only)
- No deaths
- Pacifist (only applied on the final level if the player has killed zero enemies across the whole save)

This should encourage players to return to the level and get a higher score, potentially seeing the flower planting system. The score will be displayed in a room, and the player can interact with each of the elements to show the score for each category.

## Dialogue System

While it should not be used extensively as it may break the flow of the game, this system should be complex enough to handle all dialogue requirements it needs.

It must:

- Allow for one to easily add multiple languages
- Allow for timing cues
- Allow for branching options
- Allow for quest completions
- Allow for different dialogue based on external events
- Allowed to change variables
- Prompting characters to face different characters
- Allow for changing text colour (this should allow for accessibility changes)

## Death Screens

To encourage the player to use healing items and use better movement to avoid taking damage, too much damage will result in the player's death, and a screen will need to pop up allowing the player to load the most recent save and potentially showing a useful tip on the screen to avoid death the next time. The screen could also give other statistics such as how many the enemies had killed.

## Graphical User Interface (GUI)

Many users are not exceptional at navigating and using command-line interfaces, and the human brain can process images faster than it can process words, and using the mouse is more accessible than using text arguments. As such, I will be implementing both a GUI for menu navigation, and a Heads-up Display (HUD), which will communicate information to the player while playing the game. Some elements of the HUD may be in text and numbers, while other will be in graphical solutions such as bars and images.

## Limitations

Within every project, there are limitations that one must consider, since not everything that one would like to do can be accomplished. Sometimes, trying to implement everything you want will in fact ruin the final project. Instead of trying to break all the constraints, one must instead make considerations and work around these considerations. If possible, some of the limitations could be considered challenges and

you should work to break it down and produce a better product, but the costs of doing so must be accurately estimated and the reasons must be weighted accurately against the problems it may cause.

Limitation	Why?
Time	Time is the most obvious limitation – the never-changing constraint that will limit how far I can progress with the game. With infinite time, everything in the game could get infinitely refined, and many features could be added to the game. However, infinite time is not something we have the luxury to. In fact, the entire project must be done within the space of 8 months. Therefore, there will be features that would be desirable but can't be implemented and some features which will not be as refined as preferable.
Budget	The budget for this entire project is nothing. This is a severe limitation meaning that nothing can be outsourced, and while usually you could hire better programmer, and an entire sound design team, nothing of the sorts can be done here. This will have repercussions for the software limitation as well and means that if there are any textures or models I want to use, they must be free or made by myself.
Hardware	Maintaining the performance of the game is very important because not everyone has access to top-notch hardware, and the game must run extremely well since speedrunners wish to have zero-latency from sending their actions to receiving feedback on the screen. Therefore, the game must be made as simply as possible, with as much optimisation as well. However, I am also unable to test on different types of hardware, and so the best I can do is testing on Windows on my own system. Furthermore, the game will be only for the PC, since I do not own a console, nor can I get the game files onto a console if I owned one.
Software	Software is also a constraint since I can't acquire all the professional applications, and some applications I can't run due to my own hardware. Since the budget is zero, all software and third-party resources must all be free, or hand made.
Proficiency	While I have used Unity and C# in the past, they were all for basic projects that were copied from tutorials, and as such I will need to learn more of the quirks of the engine and its language, which will take even more of my time to build the game.

## Solution Requirements

While it would be ideal to have all of the essential features included in the game, not every single one needs to be implemented to the same level of usability as others.

Based on the limitations, some features may need to be cut from the game, as is typical within the industry.

Requirement	Justification
Movement and Combat	Both movement and combat are single-

	handedly the most important requirements to exist for this NEA, since they define the primary genre of the game. Without either of these features, it would be incorrect to call the game a movement-shooter.
Saving	Every single modern game that is released nowadays has saving since no player is ever expected to finish an entire game in one sitting. To maintain the player's retention, they must be able to return to most of their progress still available to access.
GUI	A GUI is integral for the user to be able to interact with the game, since many engines do not allow the player direct access to a console/terminal for a command-line-interface. A CLI is also far more difficult to use and is less accessible to the average gamer, making this a requirement of the final project solution.
Death Screen	This also includes the player being able to die, and this is incredibly important for gameplay. Death is the risk factor, and it is what the player is intending to avoid as they are rushing around and firing at the other enemies. With no risk, there is no need for the player to attempt to play the game as intended. Once death has been achieved, it is important that the player is able to go back to their previous save or restart the level to continue playing. Any hints on how to avoid death are an optional extra to add to the death screen.

## Hardware and Software Requirements

Requirement	Justification
Computer Hardware	The game is intended to run quickly, with as minimal delay/lag as possible, therefore the GPU requirement is only the need for integrated graphics, or a simple graphics card such that one can have an enjoyable experience, while the CPU requirement is for a chip that can adequately support its GPU. On the side of

	system memory, the game will require a minimum of 2GB of RAM, so the system would be recommended to have 8GB of RAM, so as not to cause disk thrashing with other programs that the user may have loaded in, which only less than 2.5% of people who play games do not have access to nowadays.
Peripherals	The game will have support for a mouse and keyboard combination (kb/m), but nothing else, as it would be difficult to port all of the controls that would be needed to play the game to another appropriate input system, such as touch controls. The kb/m input scheme is also the most commonly used one, as evidenced by all the games in my research having support for it as the primary input method.
Operating System	My project will only be available upon a 64-bit Windows system since that is the most widely used type of system for playing games, coming in at 96.78% of all gamers on the Steam platform according to the Steam Hardware Survey of August 2024.
Storage	One would need as much storage as is required to contain all the files of the game and have at least a few gigabytes leftover for a swapfile/pagefile, in case the computer needs to use virtual storage.
Software	Many games use the same set of C++ Redistributables and depending on the individual's proficiency in playing games, they may or may not already have the relevant packages installed. This also includes .Net 4.0, the appropriate drivers for their device and the XNA Framework Redistributable.

## Success Criteria

Criterion	Justification
Strafing Movement	Strafing is the barebones required option for moving, allowing access to 2 dimensions, and even the third dimension when slopes are introduced.

Jump	While strafing can allow the player to move vertically, it is dependent on the level geometry, and it would instead be a better option to allow the player the ability to jump while on the ground, and maybe even a double jump while in midair.
Crouching	To avoid the enemy gunfire or to enter certain sections of puzzles in the game, crouching is a criterion as it allows the player to decrease the surface area that is exposed.
Firing, swapping, reloading, zooming different weapons	As mentioned, the player must be able to fire weapons to retaliate and survive for longer against their enemies. It is also part of the genre's title and would be fraud to use the genre 'movement-shooter' without the ability to shoot. Having different actions to perform on these weapons allows the player to use the weapons appropriately in different situations, especially with the inclusion of multiple weapon types. This would encourage the player to play differently, and experiment when their current playstyle may not perform the best.
Can take damage	While the player can damage others, in turn, they too must be able to take damage, allowing an even playing field and forcing the player to take better and safer movement routes, or prioritise more dangerous enemies.
Ability to go back to a save on death	Once the player's health reaches below 0, the only logical option is for the player to die, which reinforces the risk with taking damage. The player must then be able to continue playing which in a singleplayer campaign, is usually done by reloading a previous save. The death screen will contain a button to do so, by selecting between a variety of saves.
Saving	This allows the player to retain their current progress for later, such as when they close the game, or if they suspect that they might be in a tricky situation.
Loading	Conversely, one must be able to load the save that they made to bring meaning to the saving feature and re-enter the state

	that they were in when making the save.
Interactable menus with audio feedback	The GUI is the most accessible method of navigating through menus and interacting with the game. Out of all of these, the button is the most intuitive and useful of selecting options, instead of a scrollable menu. These menus will also have audio feedback to tell the player that their action had meaning and influenced the game.
Dialogue	Dialogue will allow the player to understand what happens on within the game world, and this needs to be both easily visible to the player, and also explanatory, while allowing the player the ability to interact back with the world.
Enemies searching and fighting back	The player will be fighting against the enemies in the world, who will be searching for the player to fight against, along with any entities that are fighting with the player. This adds some level of difficulty to the game along with the thrill of avoiding the enemies appropriately
The game runs at a minimum of 30 fps on typical hardware, as according to the Steam Hardware Survey	This makes sure that the game is not too intensive to run, and that it is also enjoyable to play. This will entice more people to play the game as possible, increases sales for SillyAustralian. The most typical hardware that people playing games are running have Windows 11 installed on them, with 16 GB of Ram and a hex-core CPU. Graphically, the majority of people have Nvidia GPUs, but this does not mean that the game should be extremely difficult to run.
The game will have as few glitches as possible, with absolutely no game-breaking ones	This makes sure that the player's experience is as enjoyable as possible, and that they are not frustrated from something that they did not expect to happen. This will retain as many players as possible on the game.

# Design

## Decomposition

### Combat

#### *Weaponry*

##### Swapping

Being able to swap out your weaponry on the fly is what encourages players to try new things in the game. This swaps the currently selected weapon with the weapon that the player is looking at.

##### Guns

1 of the 3 types of weaponry is guns that allow the player to deal damage from range, firing bullets that damage upon clicking the button via a raycast.

##### Melee

The second type is melee weaponry which deals high damage from close range, like bats and blades, requiring a collision check instead.

##### Grenades

The last type is grenades, which are lobbed, following a physical arc and dealing explosive damage based on its range.

### *Player*

#### Movement

##### *Horizontal*

All movement that uses the basic strafing keys are handled similarly within the same function, moving the player in the direction of the keys relative to the current facing angle in the horizontal plane. This will all be held in one function, that references other functions to access the relevant acceleration values for the different movement types and the speed limits.

##### *Vertical*

This encompasses jumping, gravity and the lack of control the player should have when moving in air. While the player is not on the ground, gravity will constantly accelerate them downwards. Jumping allows one to get over an obstacle by adding vertical velocity, but it is also possible to ramp off slopes and get far more verticality that way. Due to this,

one will have lower control of the character once in the air, needing them to plan their original trajectory. Ramping will be automatically handled by the physics system that is built into Unity 3D, while both jumping and gravity will be separate functions. Jumping will be triggered upon the input being pressed, while gravity will be triggered every frame.

### *Wallrunning*

Wallrunning is a special case of movement since the plane of movement then becomes that of the wall you are running on, and the player can only move forward on this wall, fall off or jump off. This allows them to stick to the wall and is an entirely separate movement option than the others.

### *Entities*

All entity types are shared between allies (friendlies) and enemies, bar the civilians which are only on the enemy side, as is in line with the setting of the game.

### Drones

An enemy that flies around and maintains a height-advantage on the player where possible. Depending on their type, they may act differently, such as shooting from afar for ranged drones, or moving towards the player for explosive and melee drones.

### Soldiers

An enemy that moves around only on the ground and will drop weapons and ammunition when killed. Will stay away from the player but will attempt to maintain line of sight.

### Tanks

Large and slow ground-based vehicles that deal heavy explosive damage occasionally. Have exceptional range compared to any other type of troop.

### Civilians

Cannot deal damage and will run away from enemies.

## Environment

### *Ground*

Allows for all basic movement and disables gravity while the player is stood upon it. It also prevents the player from falling out of the bounds of the level. Moving up a slope will convert some of your horizontal velocity to vertical velocity as per the physics dictate.

### *Walls*

The player can wallrun upon them but loses all other types of movement except jumping. Also disables gravity and prevents access to certain places.

### *Pickups*

Items will be scattered around the level, such as coins or weapons that the player can pick up and use, encouraging exploration and challenge.

### *Interacts*

Objectives for the level can be anything and are interacted with. Other things like doors and upgrades can also be interacted with.

## UI

### *HUD*

The heads-up display shows important information to the player, such as their health, what the active weapon is, how much ammunition that weapon has, and outlines for enemies for clarity. It cannot be interacted with but is a necessary part of gameplay.

### *Menu*

#### Title screen

Loads on startup, allowing access to the options menu and save menu, along with exiting the game. Displays the name of the game

### Options

Changes settings such as game volume and mouse sensitivity.

**Pause**

Pauses the game and allows the player to return to the main menu. Does not save the current state of the game.

**Save menu**

Displays all the save slots, each displaying 4 things: the slot number, the save history menu, the time spent in the save, and the name of the level.

**Checkpoints**

Will automatically save the progress of the player in the game at these points, creating an autosave.

***Load game***

Once a save has been selected, it will load the attributes into memory and load the level

**Dialogue*****Interpreter***

Reads the .dlg file and translates the instructions to instructions/lines for the manager to assign.

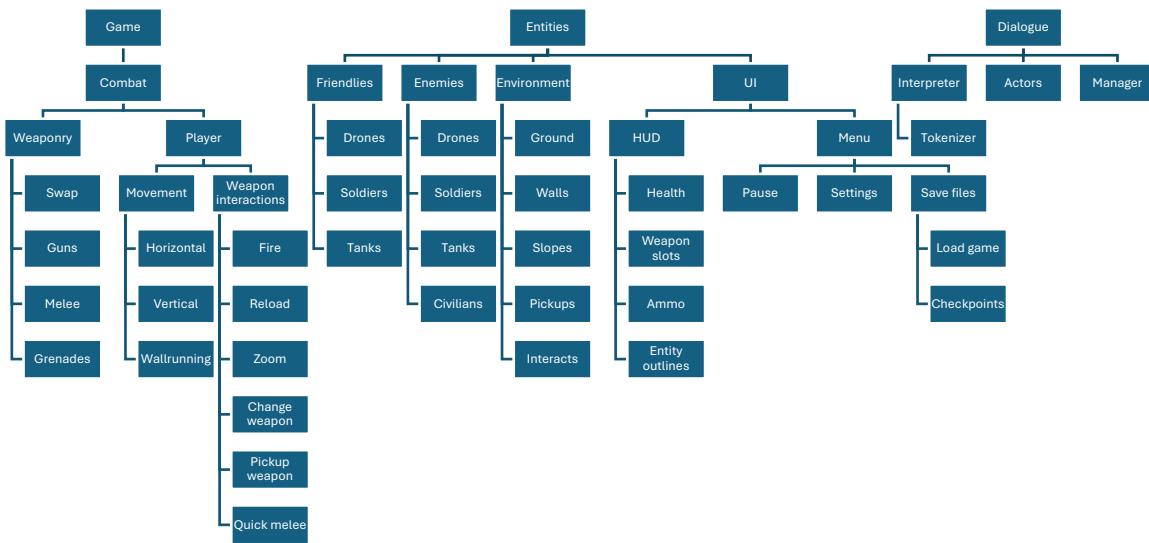
***Actors***

Will look towards appropriate actors according to the interaction and message sent by the manager.

***Managers***

Handles the receiving of instructions from the interpreter and assigning the appropriate actors their roles, or sending the subtitles/choices to the subtitles text box

## Structure Definition



## Algorithms

### Movement

```

procedure SetPlayerDimensions(height)
    playerCollider.height = height
endprocedure
  
```

This changes the dimensions of the player which will be shaped like a capsule (a cylinder with hemispheres at either end) to be set to the parameter of height, allowing for the scaling of the player character and crouching.

```

function CrouchControlState()
    holdCrouch = playerInputs.Player.HoldCrouch.inProgress
    if playerInputs.Player.toggleCrouch.triggered then
        if toggleCrouch == true then
            toggleCrouch = false
        else
            toggleCrouch = true
        endif
    endif
    if holdCrouch == true then
        lastHoldCrouchState = true
        return true
    else
        if lastHoldCrouchState == true then
            toggleCrouch = false
        endif
        lastHoldCrouchState = false
        return toggleCrouch
    endif
endfunction

```

Since the stakeholder requested both the ability to toggle crouch and a crouch that must be held using two separate keys (mentioned in the first interview), this function decides what the appropriate crouch state is, based upon the values of the keys pressed and their previous values.

```

procedure Gravity()
    if NOT isGrounded then
        movement = movement - gravity * Time.deltaTime
    endif
endprocedure

```

So long as the player does not have contact with the ground, they feel a constant acceleration of gravity downwards - a necessary factor of a movement-shooter game where the intuitive physics is key.

```

procedure Crouch()
    if lastMovementState != movementState then
        if movementState == PlayerMovementState.crouching OR
movementState == PlayerMovementState.sliding then
            SetPlayerDimensions(crouchingHeight, playerRadius)
        else
            SetPlayerDimensions(standingHeight, playerRadius)
        endif
    endif
endprocedure

```

When the player has just changed their movement state, and that state is from standing to crouching, or vice versa, it calls the relevant function with the correct dimensions. Having this be a simple function reduces the chances for any bugs which fulfils the primary requirement for Amarveer Flora, and also makes it highly maintainable, simultaneously participating to the needs of Henry Masters.

```

procedure Movement()
    inputDirection =
playerInputs.Player.Movement.ReadValue<Vector2>().normalized
    if inputDirection.magnitude != 0
        if movementState != PlayerMovementState.boosting
            movementState = PlayerMovementState.walking
        endif
        float speedMultiplier = CalculateAccelerationMultiplier()
        Vector3 multipliedVelocity = new
Vector2(inputDirection.x, inputDirection.y) *
movementAcceleration * speedMultiplier * Time.deltaTime
        Vector3 directedVelocity = new Vector2(inputDirection.x,
inputDirection.y) * movementAcceleration * 20 * Time.deltaTime
        float alignment = Vector3.Dot(player.velocity.normalized,
directedVelocity.normalized)
        Vector3 lerpedVelocity = Vector3.Slerp(directedVelocity,
multipliedVelocity, alignment)
        player.AddForce(new Vector3(inputDirection.x, 0,
inputDirection.y) * movementAcceleration * speedMultiplier *
Time.deltaTime, ForceMode.Impulse)
    endif
endprocedure

```

This is the main movement code, where it slowly attempts to align the player's velocity with the current movement being slowly aligned to face towards the direction that the

player intends to move towards. It only does this if the player wishes to move, otherwise it can skip out on all the processing and let the friction reduce the player's speed naturally. This way, all the movement, both horizontal and vertical is all applied simultaneously and only when necessary and does not cause any errors with the physics system that Unity uses which contributes to Amarveer Flora's primary need of as little bugs as possible.

```
procedure Jump()
    direction = Vector3.up * jumpForce
    if velocity.y < 0 then
        direction.y = direction.y - velocity.y
    endif
    if isGrounded == true then
        movement = movement + direction
    else if airJumpsLeft > 0 then
        movement = movement + direction
        airJumpsLeft = airJumpsLeft - 1
    endif
endprocedure
```

One of the other primary movement options that the stakeholders requested was jumping, which is also vital in a game centred around movement. This code also handles the ability to double jump, and any potential amount of jumps the player could have by having 2 variables – airJumpsLeft tracks how many more times the player is allowed to jump in the air before they must touch the landscape of the level. Once they do collide with the level geometry, then airJumpsTotal resets airJumpsLeft to the appropriate value so that the player can restart jumping in mid-air. This is handled by the following code:

```

procedure CollisionDetected(collision)
    if collision.contacts.Length > 0 then
        for i = 0 to collision.contacts.Length - 1
            slopeAngle = Vector3.Angle(contact.normal,
Vector3.up)
            airJumpsLeft = airJumpsTotal
            if slopeAngle > maxSlope then
                wallNormal = contact.normal
                Wallrun()
            else
                isGrounded = true
                groundNormal = contact.normal
            endif
            next i
    else
        isGrounded = false
        groundNormal = Vector3.up
    endif
endprocedure

```

This handles all responses to collisions, such as deciding if the surface touched is a wall or walkable floor and setting isGrounded to true for the gravity calculations. This is better than this other algorithm (see below), which has to make an extra check on the physics system, rather than leveraging the automatic function calls it makes. It also only checks if one is on the floor, rather than doing 2 jobs with the same checks.

```

function GroundCheck()
    return Physics.CheckSphere(new
Vector3(player.transform.position.x, player.transform.position.y
- groundDistance + (0.99f * player.transform.localScale.x *
playerCollider.radius) - (player.transform.localScale.y *
playerCollider.height)/2, player.transform.position.z),
player.transform.localScale.x * playerCollider.radius * 0.99f,
playerLayer)
endfunction

```

This function creates a sphere that has the same radius as the player and is positioned at the player's feet, checking if there is anything intersecting that sphere. This is more expensive than the above solution and so will not be used.

```

procedure Boost()
    movementState = movementState.boosting
    inputDirection = playerInputs.Player.Movement.ReadValue()
    boostMovement = player.rotation * inputDirection *
    groundVelocity.magnitude * (boostMultiplier - 1)
    movement = movement + boostMovement
endprocedure

```

An optional feature that the founder of SillyAustralian wanted included, boost takes the current player's velocity and multiplies it by a factor, another potentially upgradeable value for the player. It then adds this value onto the player's current velocity, meaning it could be used by a player to instantly turn around and move in the other direction.

```

procedure Look()
    lookMotion = lookAction.ReadValue<Vector2> * mouseSensitivity
    / 20
    playerBody.rotation = playerBody.rotation *
    Quaternion.Euler(Vector3.up * lookMotion.x)
    // Apply the vertical rotation to the camera
    verticalRotation = verticalRotation - lookMotion.movementY
    verticalRotation = Mathf.Clamp(verticalRotation, -90, 90)
    transform.localRotation = Quaternion.Euler(verticalRotation,
    0, 0)
endprocedure

```

The final movement algorithm is the one that handles the player's ability to look around in 3-dimensions. It converts the player's horizontal mouse movement to a horizontal rotation and their vertical movement into an up-down looking motion.

## Enemy Entities

```

procedure EnemyLoop()
    velocity = rigidbody.linearVelocity
    if state != enemyState.None then
        if target != null then
            RotateToTarget(target.transform)
        endif
        if state == enemyState.Combat then
            Move()
            Combat()
        else if state == enemyState.Searching then
            if (searchingTarget -
                rigidbody.transform.position).magnitude < 1 || NOT
                searchTargetSet then
                SelectTarget()
                searchTargetSet = true
            Move()
            Searching()
        endif
    endif
endprocedure

```

This controls the enemies next actions, depending on the state of the enemy. The enemy shall have 3 states that it references within its code: None, Searching, and Combat. None is the state that shall be used to deactivate the enemy's AI when it is not in use, such as before a combat scenario, while Searching is for when the AI is active, and looking for enemies. Finally, Combat is for the time once an enemy has been discovered, and hence the AI should fight it. To decide when the entity has reached the target, they just need to reach within 1 metre of it as this will make sure that it is not getting stuck on any floating-point errors.

Movement by enemy entities will use a similar algorithm to the player's movement and as such will not be re-detailed here.

```
procedure Searching()
    for i = 0 to rayCount - 1
        hitInfo = new hitInfo()
        rayDirection = Quaternion.AngleAxis(-45f + rayAngle * i,
Vector3.up) * Vector3.forward
        rayDirection = rigidbody.transform.position *
rayDirection
        if Physics.Raycast(rigidbody.transform.position,
rayDirection, hitInfo:byRef, DetectionRange, playerLayer) then
            state = enemyState.Combat
            UpdateManager(enemyState.Combat)
            target = hitInfo.collider.gameObject
            break
        endif
    next i
```

The enemy searches via shooting multiple rays out from its front within 45 degrees either side of the direction it is facing in. It sees what is a slit of vision and if the player collides with any of the rays, then the player is set as the target, changing the mode to combat.

```

procedure SelectTarget()
    found = false
    point = GetClosestPoint(found:byRef)
    if NOT found then
        searchingTarget.y = rigidbody.transform.position.y -
DetectionRange
        return
    endif
    entityToPoint = rigidbody.transform.position - point
    if entityToPoint.magnitude > DetectionRange then
        searchingTarget = point - entityToPoint.normalized *
Random.Random(0, DetectionRange)
        return
    endif
    // Check the enemy can reach the target
    accessible = false
    while NOT accessible
        searchingTarget.x = Random.Range(-DetectionRange,
DetectionRange)
        searchingTarget.y = Random.Range(-DetectionRange,
DetectionRange)
        searchingTarget.z = Random.Range(-DetectionRange,
DetectionRange)
        searchingTarget = searchingTarget +
rigidbody.transform.position
        hitInfo = new RaycastHit()
        if Physics.Raycast(rigidbody.transform.position,
(searchingTarget - rigidbody.transform.position).normalized,
hitInfo:byRef, DetectionRange) then
            if (hitInfo.point -
rigidbody.transform.position).magnitude < 1 then
                continue
            else
                searchingTarget = hitInfo.point
            endif
        endif
        accessible = true
    endwhile
endprocedure

```

Selecting an appropriate target for the enemy to move to while it is searching is a difficult procedure. One option is to make the enemy to move along predefined paths,

but for better gameplay interaction, the enemy itself chooses a random point that it can see and moves towards this target. The target is confirmed to be within the enemy's vision by firing a raycast from the enemy to the target and confirming that it does not interact with anything.

```
procedure RotateToTarget(target)
    targetDirection = target.transform.position -
    rigidbody.transform.position
    dirInQuaternion =
    Quaternion.Euler(targetDirection.normalized)
    rigidbody.transform.rotation =
    Quaternion.Slerp(rigidbody.transform.rotation, dirInQuaternion,
    RotationSpeed * Time.fixedDeltaTime)
endprocedure
```

The final algorithm that the enemy needs to use is a method to rotate towards its target, so that it is facing the right way when it needs. This is done by spherically interpolating between the direction towards the target and the current direction of the enemy. This makes the enemy move in a manner more visually appealing, rather than snapping towards the target.

### Weapon and its Interaction

```
procedure Fire()
    Instantiate(bullet, barrel.transform.position,
    barrel.rotation)
endprocedure
```

Arguably the easiest function in the entire game, it fires in a bullet by spawning in a new bullet. Nothing can go wrong here.

```

procedure BulletCheck()
    RaycastHit hitInfo = new RaycastHit()
    if Physics.Raycast(
        origin: transform.position,
        direction: transform.rotation * Vector2.up
        hitInfo: hitInfo:byRef
    ) then
        hit = hitInfo.collider.gameObject
        entityClass = hit.GetComponent<Entity>()
        if entityClass != null then
            hit.SendMessage("Damage", damageAmount)
    endif
endprocedure

```

On the other hand, the bullet itself makes sure that it hits a target and that it deals damage to a target. It does so by checking if there is an object that can be damaged in its path and hence telling it to damage itself with SendMessage().

```

procedure SwapWeapon(inputType)
    if slotsFilled > 0 then
        direction = inputType.ReadValue<int>();
        CurrentWeapon(activeWeaponSlot).SetActive(false)
        CurrentWeapon(activeWeaponSlot).transform.position =
        weaponStow.position
        activeWeaponSlot = Wrap(activeWeaponSlot + direction,
        weaponSlots)
        if SelectFireWeapon() == null then
            activeWeaponSlot = Wrap(activeWeaponSlot - direction,
            weaponSlots)
        endif
        CurrentWeapon(activeWeaponSlot).transform.position =
        weaponHold.position
        CurrentWeapon(activeWeaponSlot).SetActive(true)
    endif
endprocedure

```

Swapping a weapon requires checking if there are any weapons to swap to in the first place, but once checked all that must happen is that the current weapon is disabled and moved out of the way, and the new one is moved into the right spot and then enabled. CurrentWeapon() is a function that returns the appropriate weapon based upon the slot. This is because all the weapons are stored as individual variables and referred to via a switch statement.

```

function GetSwapSlot(wasUsed:byRef)
    if slotsFilled < 4 then
        wasUsed = false
        return activeWeaponSlot + 1
    endif
    wasUsed = true
    return activeWeaponSlot
endfunction

```

This function attempts to retrieve the slot that the player needs to access to pick up a weapon. If they do not have all their slots filled, then the next one is supplied. Otherwise, the current one is returned so that it can be swapped out and changed for the new weapon.

```

procedure Interact(inputType)
    RaycastHit hit = new RaycastHit()
    if Physics.Raycast(
        origin: playerHead.transform.position,
        direction: playerHead.transform.rotation *
    Vector3.forward,
        hitInfo: out hit
    ) then
        if hit.collider.gameObject.tag == "Weapon" then
            AddWeapon(hit.collider.gameObject)
        endif
    endif
endprocedure

```

Interact() allows the player to sense and pick up a weapon and is required due to the input system. Sensing is done via a raycast from the centre of the player's vision. And checking the hit object's tag.

```

procedure AddWeapon(hit)
    wasUsed = false;
    activeWeaponSlot = GetSwapSlot(wasUsed)
    if wasUsed then
        CurrentWeapon(activeWeaponSlot).transform.SetParent(null,
true)

        CurrentWeapon(activeWeaponSlot).transform.position =
weaponHold.position
        CurrentWeapon(activeWeaponSlot).transform.rotation =
weaponHold.rotation
    else
        SetWeapon(Wrap(activeWeaponSlot + 1), hit)
    endif

    CurrentWeapon(activeWeaponSlot).transform.SetParent(playerHead.tr
ansform, true)
    CurrentWeapon(activeWeaponSlot).transform.position =
weaponHold.position
    CurrentWeapon(activeWeaponSlot).transform.rotation =
weaponHold.rotation
endprocedure

```

To pick up a weapon, first, you must check if there is a weapon you must get rid of, which has to be enabled and unparented from the player. The picked-up weapon is parented to the player and then placed into the holding spot.

## Dialogue

```

procedure PreprocessSections()
    fileLines = dialogueFile.text.Split("\n")
    for i=0 to fileLines.Length - 1
        line = fileLines[i]
        if line.StartsWith("[") AND line.EndsWith("]") then
            dialogueSections.Add(line[1..^1], i)
        endif
    next i
endprocedure

```

Dialogue will consist of sections which the text can jump to, and these need to be preprocessed beforehand by scanning through every single line of the text file and searching for lines that both start and end with “[“ and “]”, which is the markers for the dialogue. These lines are added to a dictionary for fast lookup.

```
function RequestNext(isDialogue, language, lineID)
    connection = isDialogue ? ddm.OpenDialogueDb() :
ddm.OpenActorDb()
    lineRequest = connection.CreateCommand()
    lineRequest.CommandText = $"SELECT {language} FROM
{(isDialogue ? "Dialogue" : "Actor")} WHERE id={lineID}"
    response = lineRequest.ExecuteReader()
    response.Read()
    line = response.GetString(0)
    connection.Close()
    return line
endfunction
```

Similarly, the object needs to request a new line by requesting the appropriate ID from a database. All the above code are required to send an SQL query to a database.

```
function ConvertLine(line)
    copy = line
    firstItalics = true
    firstBold = true
    for i=0 to line.Length
        if line[i] == '\\\' then
            i++
        else if line[i] == '<' then
            hexColour = line.Substring(++i, 7)
            copy += "<color=" + hexColour + ">"
            i += 6
        else if (line[i] == '>') then
            copy += "</color>"
        else if (line[i] == '_') then
            if firstItalics then
                copy += "<i>"
                firstItalics = false
            else
                copy += "</i>"
                firstItalics = true
            endif
        else if line[i] == '*' then
            if firstBold then
                copy += "<b>"
                firstBold = false
            else
                copy += "</b>"
                firstBold = true
            endif
        else
            copy += line[i]
        endif
    return copy
endfunction
```

Parsing the retrieved line from the database is required to put it into a format read by Unity 3D, which uses RichText to display its text in UI elements. It simply runs through every character in the line and replaces it with the appropriate tag. For tracking whether a piece of text is bolded or italicised, a boolean is used for each individually.

```

procedure ReadNext(line, lineTree)
    while NOT line.EndOfStream
        char token = ReadChar(line)
        if token == '#' then
            ReadID(line, lineTree)
        else if token == '@' then
            ReadIf(line, lineTree)
        else if token == '-' then
            ReadJump(line, lineTree)
        else if token == '|' then
            ReadChoice(line, lineTree)
        else if token == ':' then
            token = ReadChar(line)
            if token == ':' then
                ReadSeparator(line, lineTree)
            endif
        else
            throw new System.Exception($"Tokenizer: The starting
character '{token}' was not recognised.")
        endif
    endwhile
endprocedure

```

ReadNext() is the main bulk of the tokenizer, identifying what the next token should be, and then passing it on to the relevant function to create that token.

```

procedure ReadID(StreamReader line, TreeNode lineTree)
    char character = ReadChar(line)
    int id = ReadNumber(line)
    TreeNode idNode = new TreeNode()
    lineTree.AddChild(idNode)

    if character == 'A' then
        idNode.AddType("actor")
    else if character == 'L' then
        idNode.AddType("line")
    else
        throw new Exception($"Tokenizer: Unable to decide if this
is an actor or line reference: {character}")
    endif
    idNode.value.Add("id", id.ToString())
endprocedure

```

This is an example of one of the token functions, first reading more characters to make sure that it indeed is a token that it can read, and throwing an error if it cannot read it. It also creates a node with right type for later parts of the interpreter to use and then adds this to the current tree.

## Save System

```

procedure Save(data, type, saveSlot)
    // Find the appropriate save number
    type = new CultureInfo("en-US",
false).TextInfo.ToTitleCase(type)
    path = Path.Combine(Application.persistentDataPath, "Saves",
"Slot" + saveSlot.ToString(), type)
    fileMatch = "Slot" + saveSlot.ToString() + type + "*.dat"
    array matches[] = Directory.GetFiles(path, fileMatch)
    Array.Sort(matches)
    fileNum = 1
    if matches.Length > 0 then
        lastFile = matches[^1]
        lastFile =
Path.GetFileNameWithoutExtension(lastFile).ToString()
        fileNum = Int32.Parse(Regex.Match(lastFile,
pattern).ToString()) + 1
    endif
    path = Path.Combine(path, "Slot" + saveSlot.ToString() + type
+ fileNum.ToString() + ".dat")
    // Save the data as json
    stream = new StreamWriter(path)
    json = JsonConvert.SerializeObject(data)
    stream.Write(json)
    stream.Close()
    // Correct the save lookup with the newly created file
    lookup = GetLookup()
    lookup[$"Slot{saveSlot}"] = path
    keys = lookup.Keys
    values = lookup.Values
    lookupWriter = new
StreamWriter(Path.Combine(Application.persistentDataPath,
"SaveLookup.txt"))
    for i=0 to key.Count - 1
        lookupWriter.WriteLine(keys[i] + "," + values[i])
    next i
    lookupWriter.Close()
endprocedure

```

The most important function for this system is the ability to save, consisting of 3 major steps: finding the appropriate number for the save; saving the data; modifying the lookup. Finding the right number is a difficult task as all we have is the files in the

directory. However, since all files are named the same way, and I can decide how these files are named, the easiest solution is to put the save number right at the end of the file's name. This way, I can use a regular expression to match the last number in the name, then convert this string to an integer and increment it to use as the new file's name. This also keeps the filename still readable for anyone trying to maintain the game and debug any problems. Next, saving the data is simple by converting the object into JSON, which is a popular format for noting information about objects in a text format. Finally, modifying just requires changing the right value in the dictionary, then writing these changes to the file.

```
function LoadSave(path)
    sr = new StreamReader(path)
    data =
JsonConvert.DeserializeObject<SaveData>(sr.ReadToEnd())
    sr.Close()

    return data
endfunction
```

Loading a save is much simpler in comparison, since the JSON just needs to be converted back into the appropriate format, which can also be done with the built-in function.

```

function ListAllSlots()
    lookupPairs = GetLookup()
    if lookupPairs == null then
        return new List<SaveSlot>()
    endif

    if NOT Directory.Exists(path) then
        Directory.CreateDirectory(path)
        return new List<SaveSlot>()
    endif

    slotList = new List<SaveSlot>()
    for i=0 to directories.Length - 1
        directory = directories[i]
        folder =
            directory.Substring(path.Length).TrimStart(Path.DirectorySeparatorChar)
        savePath = lookupPairs[folder]
        slot = new SaveSlot(savePath)
        slotList.Add(slot)
    next i
    return slotList
endfunction

```

This will be used for the menu where all the different slots are displayed, by filtering through the lookup and grabbing the most recent files in all of them. This is then used to form temporary containers for the most important information about every save.

## Usability

### Game Design

For the designs of certain sections, these have been created within Adobe Photoshop so as to make the creation of elements as fast as possible and then recreated within Unity as interactable elements. The designs were verified and improved upon by my primary stakeholder, SillyAustralian. I will be making a few designs:

- Main Menu - A simple opening menu for the player to see when they open the game. There are very few things that they need to click and hence this makes it easy to navigate.
- Save Slot Menu – Displaying all the save slots a player has; the main centrepiece will be each slot and so the player will naturally be attracted to it.

- Game HUD – Containing the most important information that the player needs to always be aware of. This will not have any interactable elements but will make it as easy as possible for the player to find this information.
- Choice – A pop-up menu with only interactable elements displaying the choices the player has to take. The player will be unable to lose their way here as everything will be in a big font.
- Subtitles – Displaying the text and lines that characters will say. This is not an important part of the screen and hence should take as little space as possible, while still being readable. This might hurt usability if it is too small and so this will have to be discussed with my stakeholder.
- Pause Menu – A menu that the player is expected to use constantly, it must be similar to the Main Menu in layout and functionality and hence will also need to be simple to use.

*First Designs:*

Main Menu:

# Abrupt Animus

Play      Options      Exit

*SillyAustralian:* This design is very simplistic and does not fit the tone of the game. Abrupt Animus is supposed to be a stark and emotionally powerful game, and the use of white for the background does not make sense in this case. Perhaps it would be better to change this to a grey, and then hence the text to white.

Save Slot Menu:

## Game Slots

- Slot 1
- Slot 2
- Slot 3
- Slot 4
- Slot 5
- Slot 6

*SillyAustralian:* Similarly with the advice for the Main Menu, the background has to be changed to fit the feel of the game. Other than that, for the save slot menu, we were hoping to implement the slots as a sideways-scrolling menu instead of a list. This way, we could display some more superficial information about each save file so the player can easily recognise which save was the one they wanted. Could you instead make the slots to be black boxes that have the slot number, the current level, and the time spent in that save?

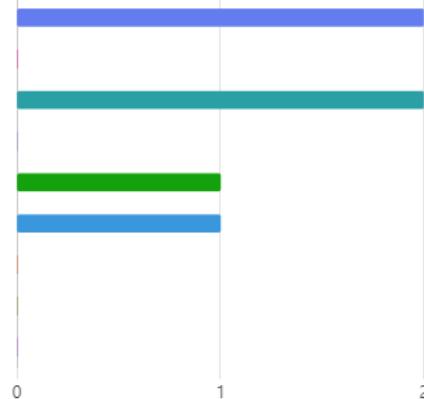
Game HUD:

For the Game HUD, since the elements have been placed in many different positions across many different games of the same genre, I decided it would be better to give both my stakeholders the form to complete and pass to some of their top executives. As such, I was able to receive 6 responses in total, which decided the position of these elements. The survey also contained any optional extra elements that could potentially be added.

1. Where should the health be on the HUD?

[More details](#)

- |                 |   |
|-----------------|---|
| ● Bottom Left   | 2 |
| ● Bottom Right  | 0 |
| ● Top Left      | 2 |
| ● Top Right     | 0 |
| ● Top Centre    | 1 |
| ● Bottom Centre | 1 |
| ● Left Centre   | 0 |
| ● Right Center  | 0 |
| ● Other         | 0 |

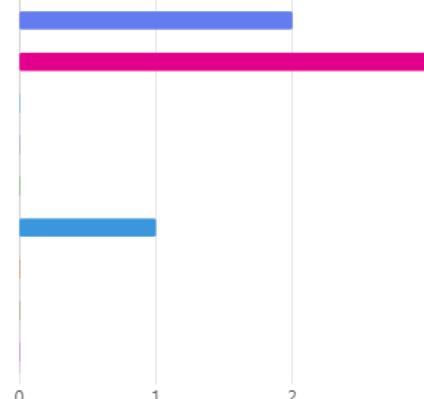


This ended with a tie between bottom left and top left, and I decided that keeping all the elements at the bottom of the screen would leave more space on the top of the screen, encouraging the player to look up and forward, and not at the ground. Hence, the health bar would be placed in the bottom left.

2. Where should the weapon slots be on the HUD?

[More details](#)

- |                 |   |
|-----------------|---|
| ● Bottom Left   | 2 |
| ● Bottom Right  | 3 |
| ● Top Left      | 0 |
| ● Top Right     | 0 |
| ● Top Centre    | 0 |
| ● Bottom Centre | 1 |
| ● Left Centre   | 0 |
| ● Right Center  | 0 |
| ● Other         | 0 |



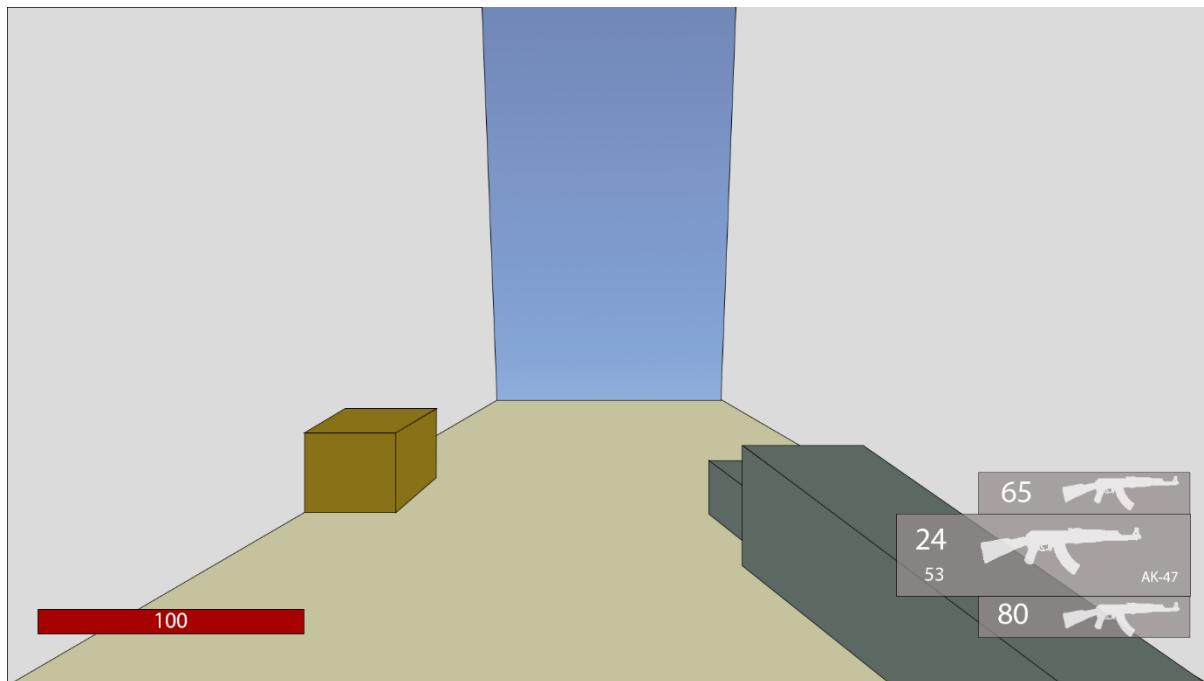
It was a clear winner that the weapon slots, meaning the currently equipped weapon and the next and previous weapons, should be placed in the bottom right, which separated the health bar and the weapon slots. This may be beneficial for the player since it does not confuse them when they need to only look at their health, which is the most important information to the player in the game.

## 3. What other things should be on the HUD?

[6 Responses](#)

ID ↑	Name	Responses
1	anonymous	total ammo in weapon
2	anonymous	Current quests/tasks,
3	anonymous	Minimap w red dots to indicate enemies
4	anonymous	ammunition
5	anonymous	A shield bar like the health bar
6	anonymous	Elements that need to be in the HUD.

As for extra elements, response 3, 5, and 6 were not helpful, since these options were not in the original request for the game, nor was there time to implement such features. Response 2 had the potential to be useful, if time permits the inclusion of a quest system, but it is highly unlikely that would be the case. Finally, response 1 and 4 both indicated that the ammunition in the weapon should also be displayed on the HUD, and as such, these features would be displayed on the HUD, resulting in the below design:



## Choice:



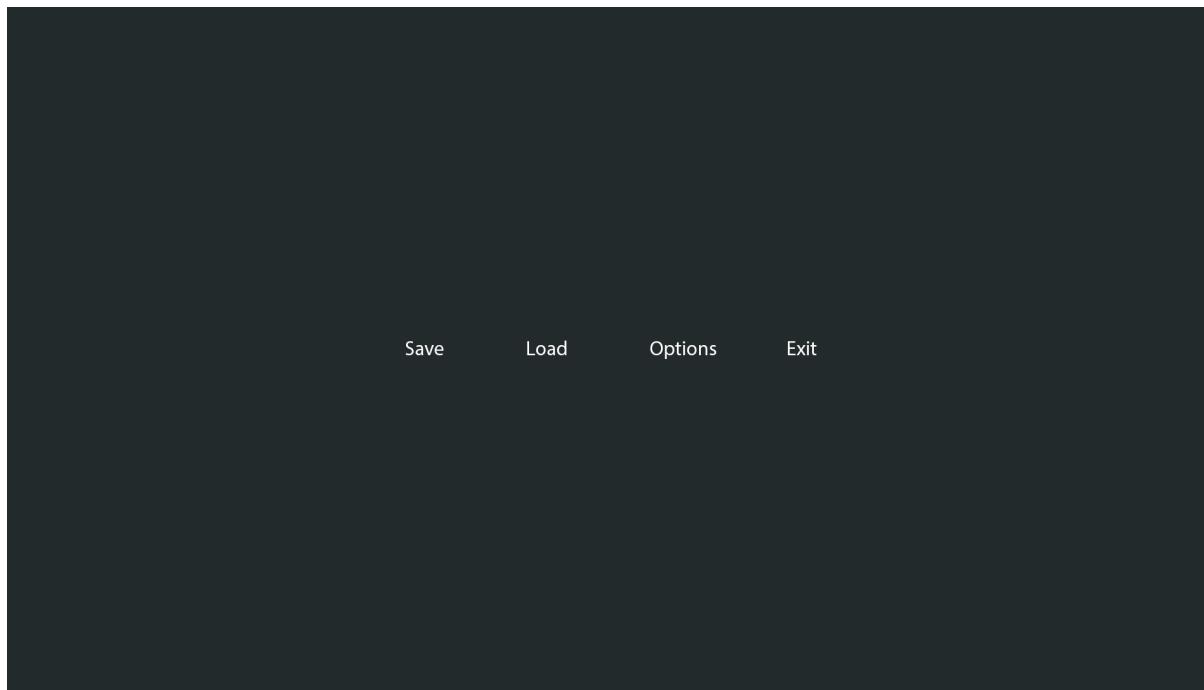
*SillyAustralian:* This design is almost ready right from the start, however, it would be better if it wasn't completely opaque, so that the player can also plan their next moves once they have finished their choice. There also needs to be a border around the choice box to clearly mark it off from the rest of the screen and make it clear.

## Subtitles:



*SillyAustralian:* The subtitles suffer from the same issues as the choice menu, and as such the same changes need to be applied.

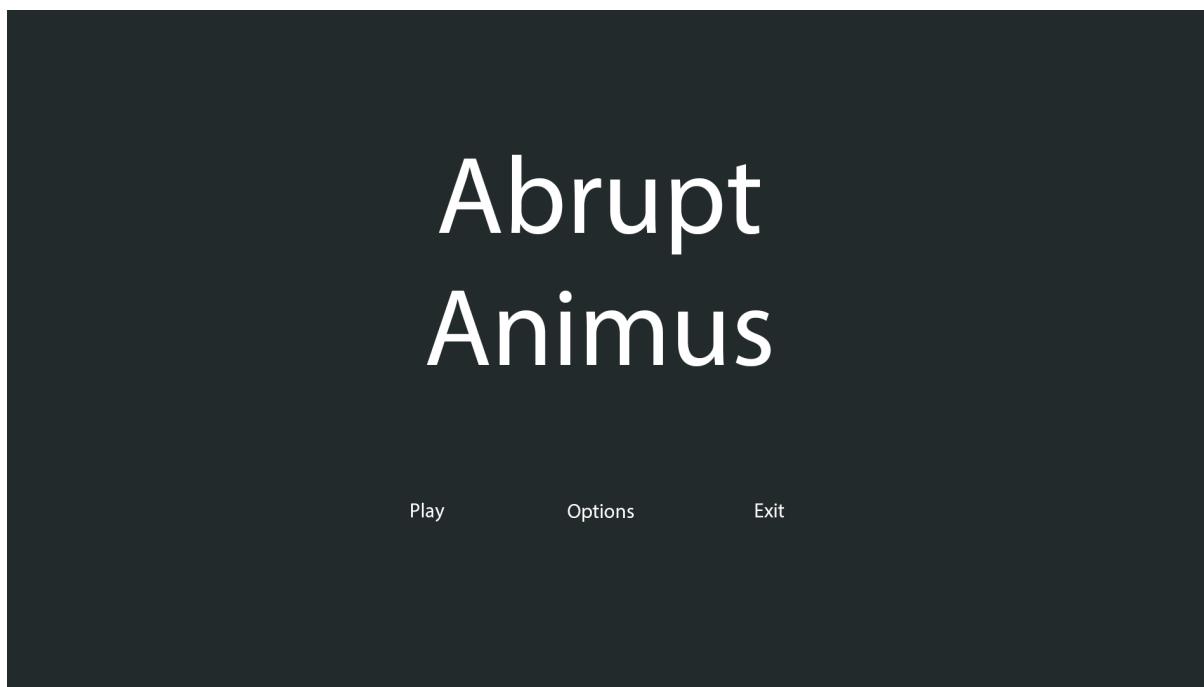
Pause Menu:



*SillyAustralian:* When in the pause menu, the player should still be able to see some of the game world, so this menu should not be a stark background, but instead it should be slightly transparent. Furthermore, this needs to have the buttons arranged vertically, with the same box background behind them.

*2nd:*

Main Menu:



*SillyAustralian:* This is much better than the previous design, however it is now difficult to see where the buttons are. Furthermore, we wish to attract the player to playing the game, not exiting it. Maybe try changing the buttons to be vertical and adding a white background to them.

Save Slot Menu:



*SillyAustralian:* The text 'Game Slots' is far too close to the actual slots, so we would like this moved upwards. Furthermore, it seems that the buttons to make a new save and load the current save are missing, so if you could add that at the bottom of the screen, that would be much better.

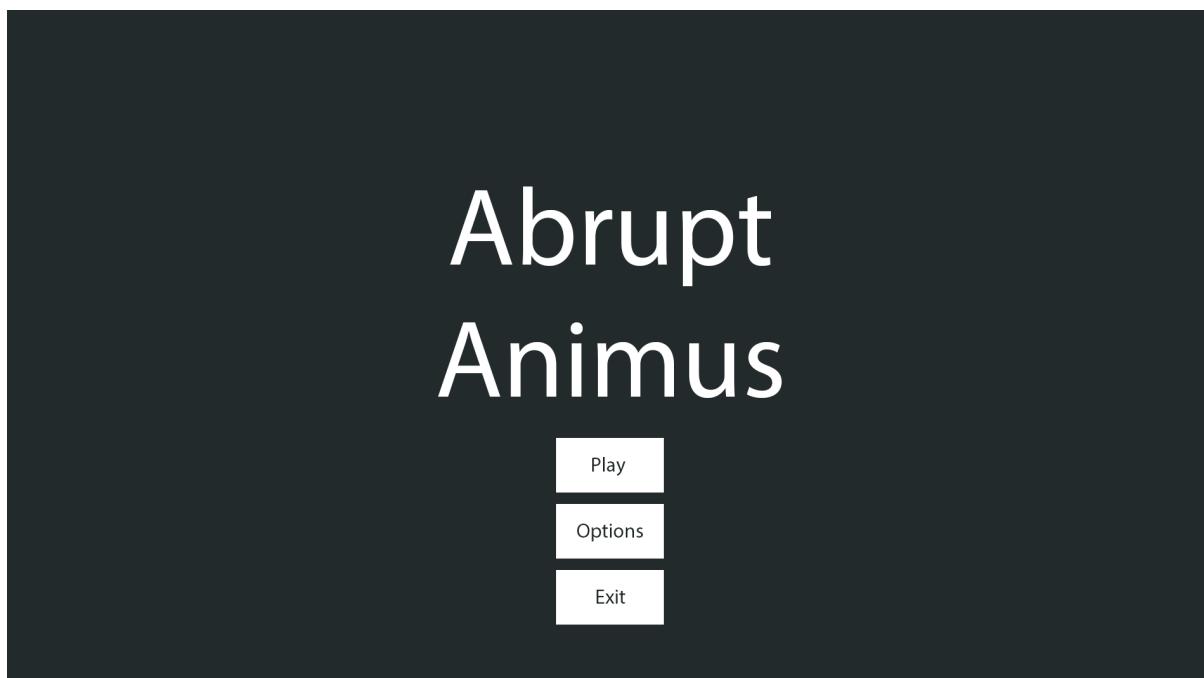
## Pause Menu:



*SillyAustralian:* The only change here that is needed is that the normal game HUD needs to be turned off when the player is in the pause menu, so only the gun and environment needs to remain.

3rd:

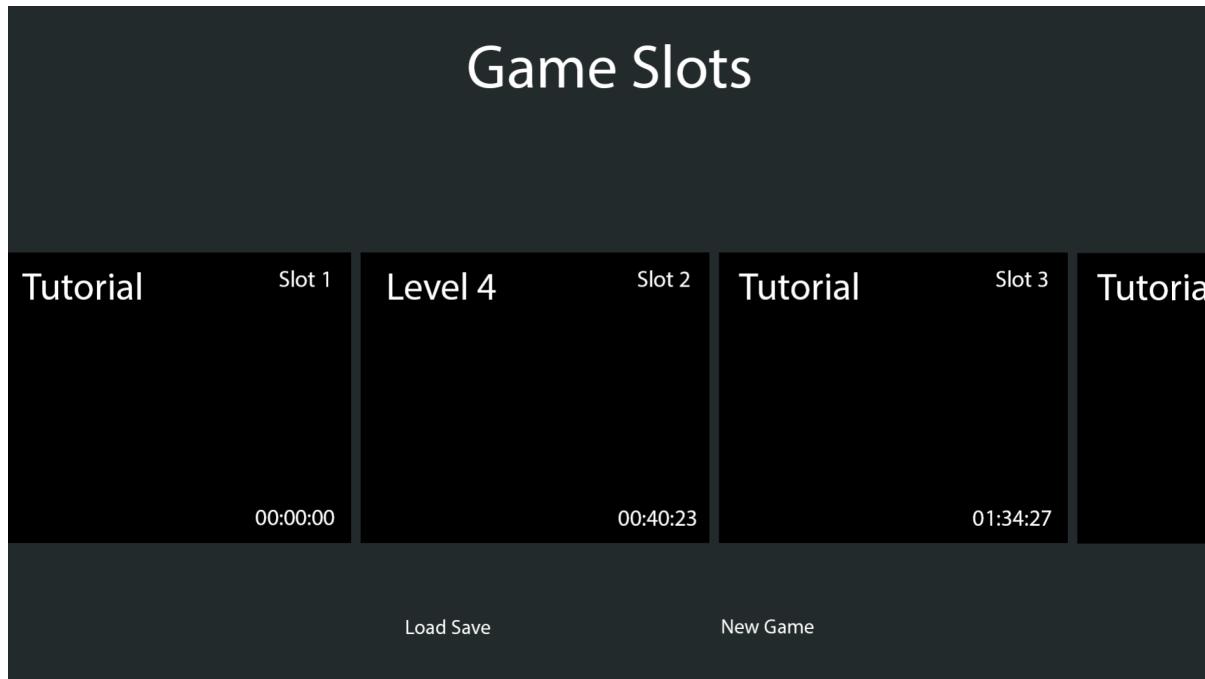
## Main Menu:



*SillyAustralian:* This is getting closer to what I had imagined in my head! Just two final points from me:

- We, as a company, have decided that we would also like the version number of the game, so could we have this in the bottom left?
- There is too much negative space at the top of the screen, so maybe we can push all the current elements upwards?

Save Slot Menu:



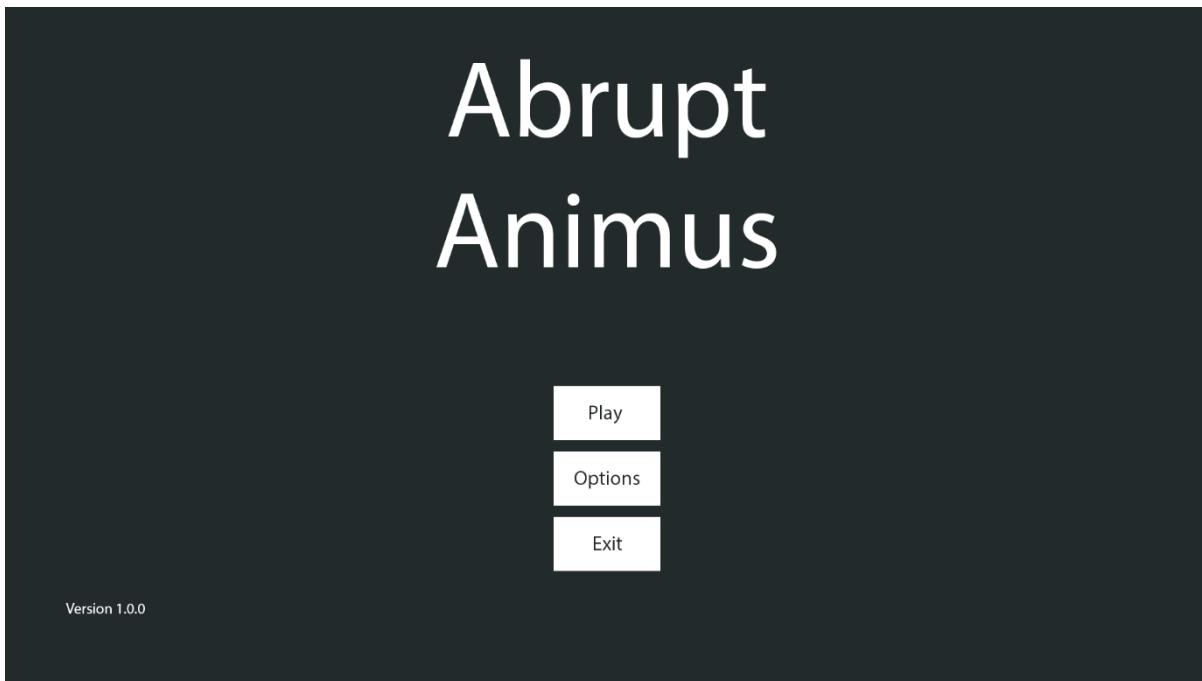
*SillyAustralian:* The buttons for the loading and creating of saves requires a background, just like the ones on the Main Menu design. There also needs to be a backdrop to the slots menu and the button panel at the bottom to make them pop out.

*Me (Eshan Fadi):* Just thinking from an implementation point of view, I've realised that there would also need to be a button for reloading the save slots, so should I add that to the button panel on the left side?

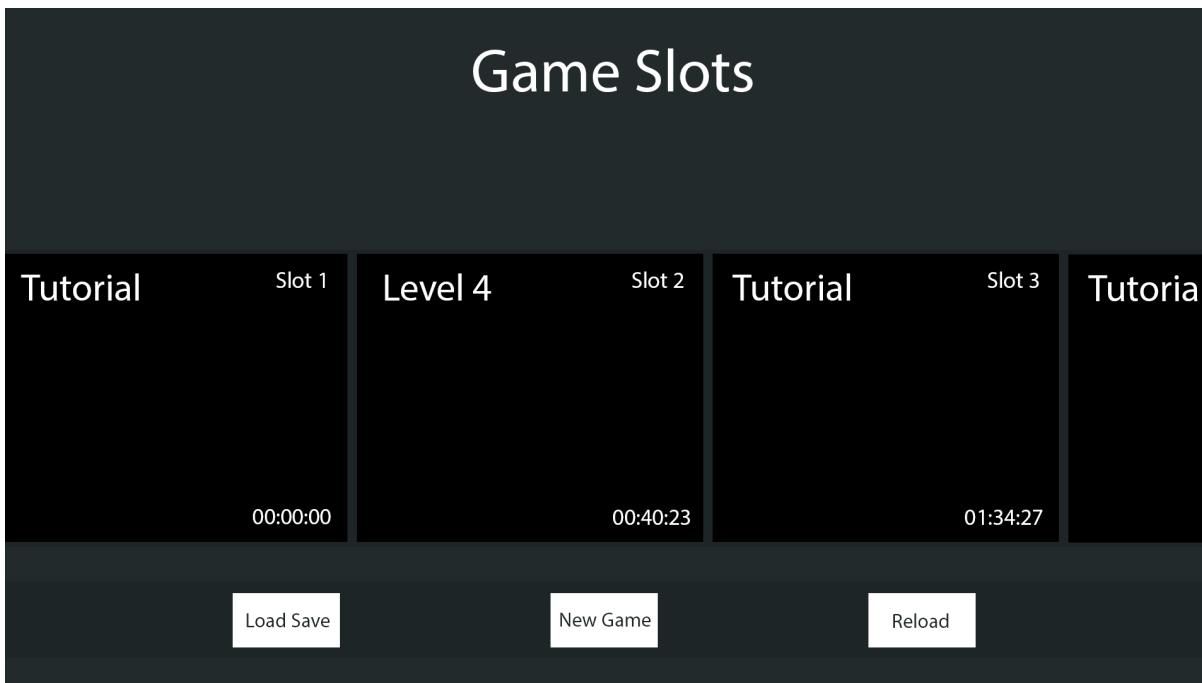
*SillyAustralian:* Hmm, I see. In that case, you should put the reload button on the right, since the 'New Game' button should be in the centre.

Final Agreed Designs:

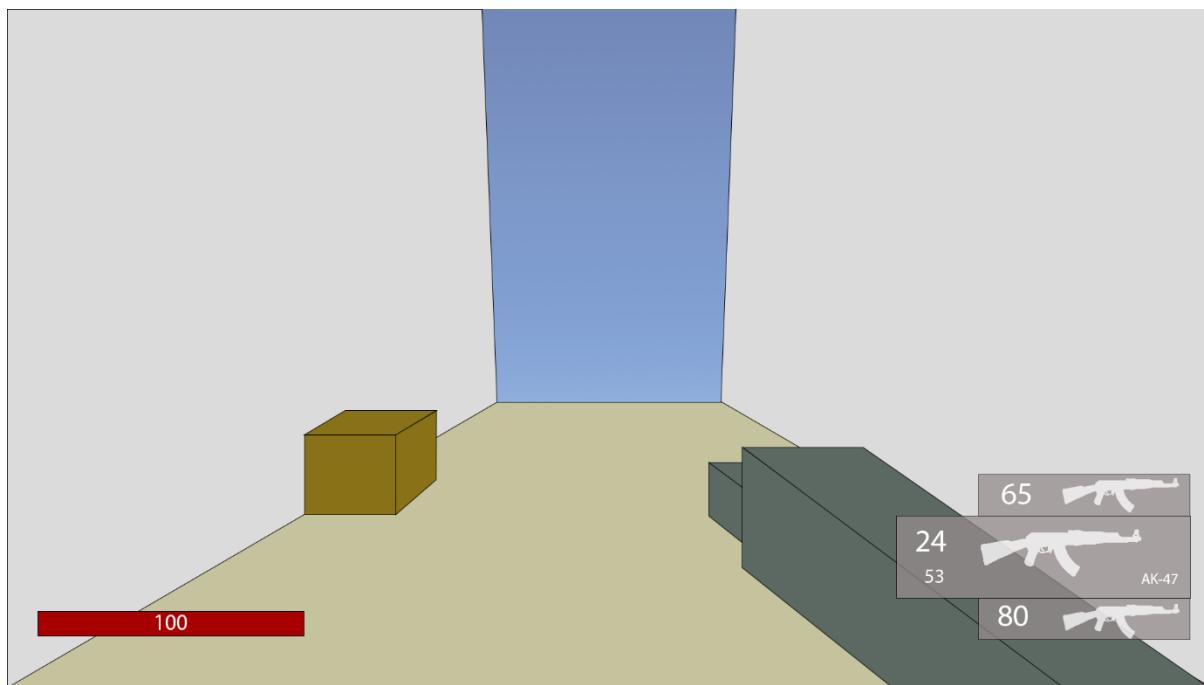
## Main Menu:



## Save Slot Menu:



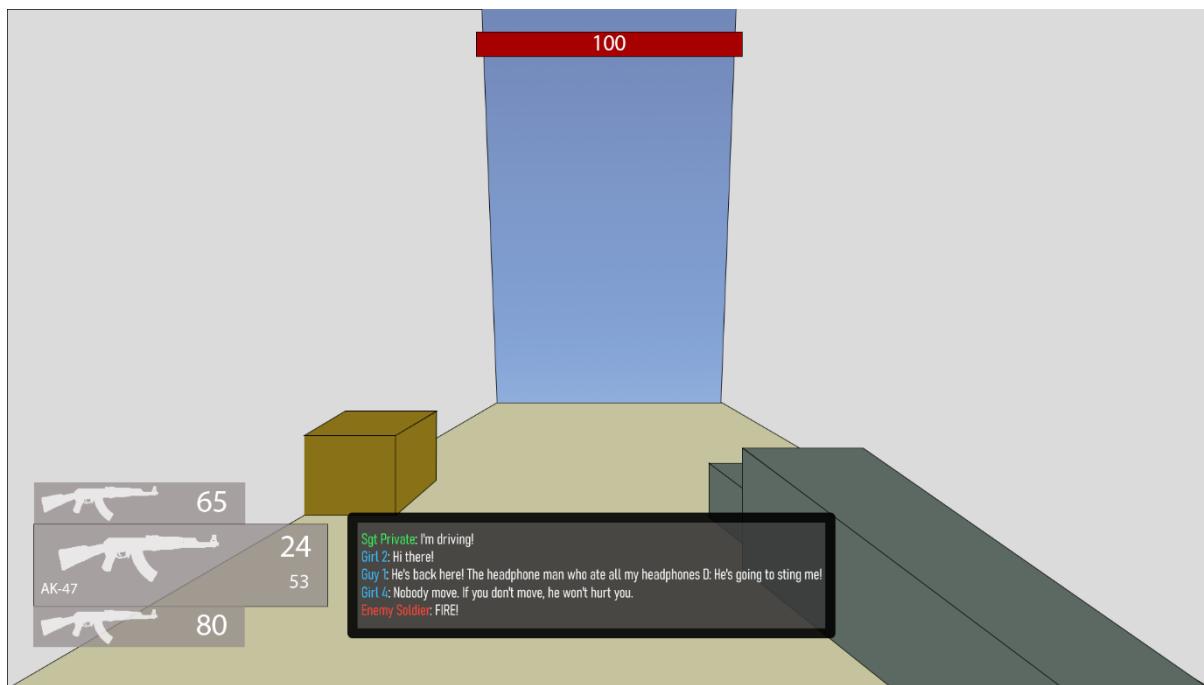
## Game HUD:



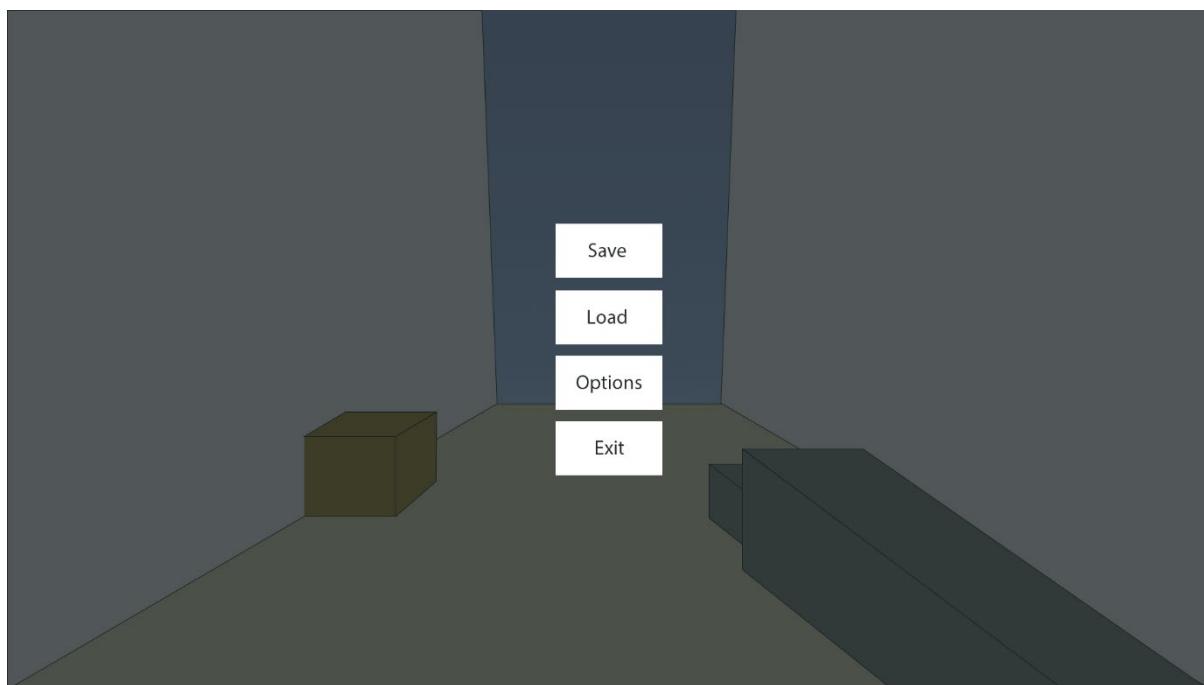
## Choice:



## Subtitles:



## Pause Menu:



## Variables and Data Structures

Type	Name	Justification
Movement		
Float	maxSpeed	Limits the player's speed so that they don't go too fast and break the game

Vector3	movement	Contains the total movement of all the different movement functions, which is then applied together at once
Float	standingHeight	The height of the player when standing
Float	crouchingHeight	The height of the player when crouched
Bool	isGrounded	Used by the gravity and jump functions to check if the player is touching the ground
Float	maxSlope	Used by the collision detection function to partition collisions into floors and walls
Float	gravity	How fast the player should accelerate downwards
Float	jumpForce	The speed the player leaves the floor when jumping
Int	airJumpsTotal	An upgradeable value that makes sure that combined with airJumpLeft, controls the ability to jump in midair
Int	airJumpsLeft	Makes sure that the player can't jump away too far
PlayerMovementState	movementState	The current state the player is in such as walking or crouching
PlayerMovementState	lastMovementState	Used by the crouching function to check whether the player has changed to crouching from a previous state to then change the player's dimensions
Float	friction	A percentage of the player's speed that is taken away per frame while on the ground
Float	drag	Same as friction but for in the air
Bool	holdCrouch	Stores the state of the key responsible for the crouch that must be held

Bool	toggleCrouch	Stores the state of the key that can toggle crouching
Bool	lastHoldCrouchState	Used to detect a change in the state of holdCrouch
Dialogue		
Database	Dialogue	A database is the best way to store similar/linked versions of the same piece of data such as different languages of text and select the right language at the appropriate moment. See below for in depth.
Tree	fileTree / lineTree	A tree is the most common method for tracking information for an interpreter, and since this interpreter does not need to track variables and the like, this can be much simpler.
Int	choice	Tracks which option the player chose in choices
Int	lineNumber	Tracks what line of the file the interpreter is on
Dictionary<string, int>	dialogueSections	Stores the appropriate sections and the lines to jump that they start on
Save System		
Text File	Save Lookup	This needs to be used to quickly load the last used save on a slot
TMP_Text	scene	A reference to the text object on the GUI that is the scene name of the save file.
Dictionary<string, string>	lookup	A usable form of Save Lookup for the code to use
List<SaveSlot>	slots	Used in the slot displaying system by the GUI after retrieving a list from the SaveSystem
Class	SaveSystem	A static class with functions handling saving the player's data, loading that data into the game,

		and listing out the saves for external systems.
Class	SaveSlot	A temporary container used for the save slot GUI with basic information about a save slot to minimise memory usage.
Entities		
Class	Entity	Handles health and damage
Class	PlayerEntity	Inherits Entity, allows the player to improve their maximum health
Class	EnemyEntity	Inheriting Entity, it implements AI searching, moving, and attacking, along with communicating with a commander.
Class	Drone	Inherits EnemyEntity, modifying functions to move in 3 dimensions instead of one the ground. Has far detection range and high speed but weak
Class	Soldier	Inherits EnemyEntity, moves on the ground with low detection range and average movement abilities and health
Class	Tank	Inherits EnemyEntity, moves on the ground with extremely far detection range and high health but extremely low speed
Float	MaxHealth	A property containing the maximum health the entity can reach
Float	Health	A property disclosing the current health the entity has
Float	DefenseMultiplier	A property that denotes how much less damage the entity takes
Float	HealingMultiplier	A property that is used to increase how much healing an entity will experience

Vector3	searchingTarget	A location for the enemy to move towards while it is searching for the player
Float	MaxMoveSpeed	A property used as a limit on the PID controller's output
Float	RotationSpeed	A property controlling how fast the entity rotates
Float	FarTargetRange	A property that is used in tandem with NearTargetRange to control midrange
Float	NearTargetRange	A property that is used in tandem with MaxTargetRange to control midrange
Float	DetectionRange	A property used to control how far the enemy can see the target
Float	proportionalGain	Used by the PID controller to control how fast the entity reaches the target
Float	derivativeGain	Used by the PID controller to control how fast the enemy slows down upon reaching the target
Float	integralGain	Used by the PID controller to resist any constant forces (like gravity)
Vector3	lastError	Used by the PID controller to track the error for calculating the derivative
Vector3	lastPosition	Used by the PID controller to track the movement for calculating the derivative
Vector3	storedIntegral	Used by the PID controller to sense any continuous errors over time
Float	maxStoredIntegral	Used by the PID controller to limit how much error is stored
Weapons		
String	WeaponType	An ID for the weapon, allowing it to be loaded by the save system
Int	TotalAmmo	The total number of shots left before more

		ammunition needs to be picked up
Int	AmmoInClip	The total number of shots left before the weapon needs to be reloaded
Class	Weapon	An abstract class that mandates a method Fire(), a weapon type, the ammunition left in a weapons clip, and the total ammunition a weapon has
Class	Gun	A ranged weapon that fires bullets, dealing damage
Class	Blade	A melee weapon using a collider to deal damage
Class	Explosive	Deals damage to all entities within its range

## Enumerators

Enumerators are more human readable way for the states of an object. Instead of having:

```
if gameState == 0 then
    print("Game has ended")
else if gameState == 1 then
    print("Game is running")
. . .
```

You could instead have an enumerator which names the states and makes it easier to read and remember. It's far easier to remember if the state is equal to running, instead of remembering that the state is equal to 5, which is running. This is hugely beneficial for maintainability which is what the primary stakeholder required – that the game was easily maintainable and could be improved by other people.

### *PlayerMovementState:*

- Idle
- Crouching
- Sliding
- Walking
- Sprinting
- Boosting
- Wallrunning

*DerivativeType:*

- Velocity
- Error

*MemberState:*

- None
- Searching
- Combat
- Dead

## Databases

This project uses a few databases throughout the game to store certain predefined constants. For saving and changing variables, other types of files are used as appropriate to the industry standard, such as level data being stored within the player's save file instead of a database.

*Dialogue:*

This database contains 2 tables that are not linked to each other through any keys: *Actor* and *Dialogue*. The *Actor* table contains only a primary ID (of type INTEGER) and the respective name of the actor to be used within the subtitles of the system. The *Dialogue* table also contains a primary ID (also of type INTEGER) and the line of dialogue in the next field. However, the field name is the name of the language that you wish to use, allowing SillyAustralian's company to add support for many other languages if they wish to, and the underlying system will need to have support for this.

*SceneInfo:*

This is a flat file containing information about every scene, such as the start position that the player should be spawned at, whether the player should be allowed to move in this scene, and what the maximum number of enemies should be, for scoring purposes. Storing the 'moveability' of the player is important, as certain scenes may be UI only, and hence the player's normal game controls should be disabled.

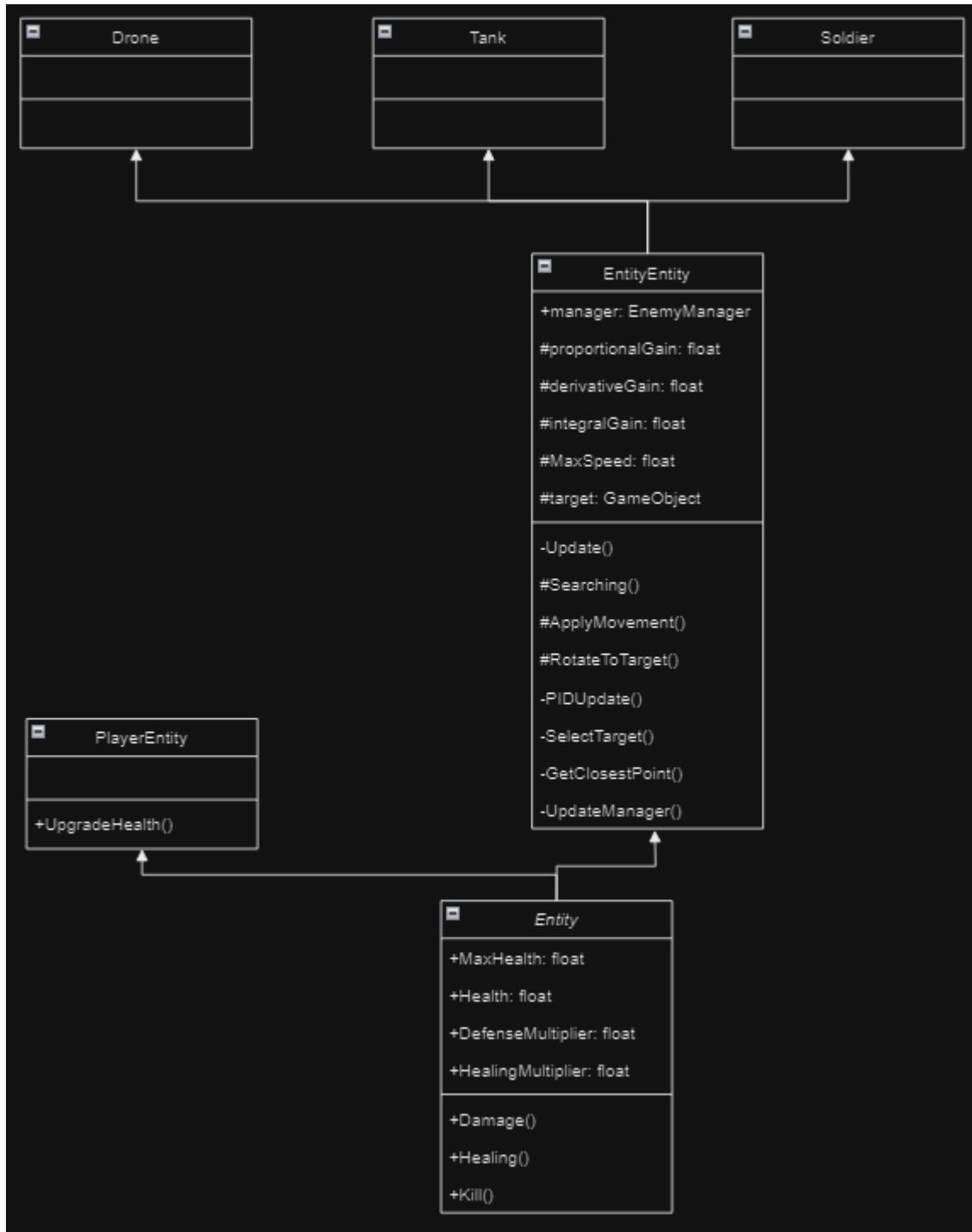
## Naming Conventions

For this project, I will be using the same methodologies for naming parts of the code. This will make the project easy to understand at a glance and will improve the maintainability for when the game is handed over to SillyAustralian.

- Variables / Class attributes – These will be written with camelCase, where the first word is in lowercase, and all subsequent words have the first letter capitalised. Examples include
- Classes / Methods / Functions / Procedures / Class properties – These will use PascalCase, which is also sometimes known as TitleCase with the spaces removed, and this is where all words in the name have the first letter capitalised.

## Class Diagrams

### Entities

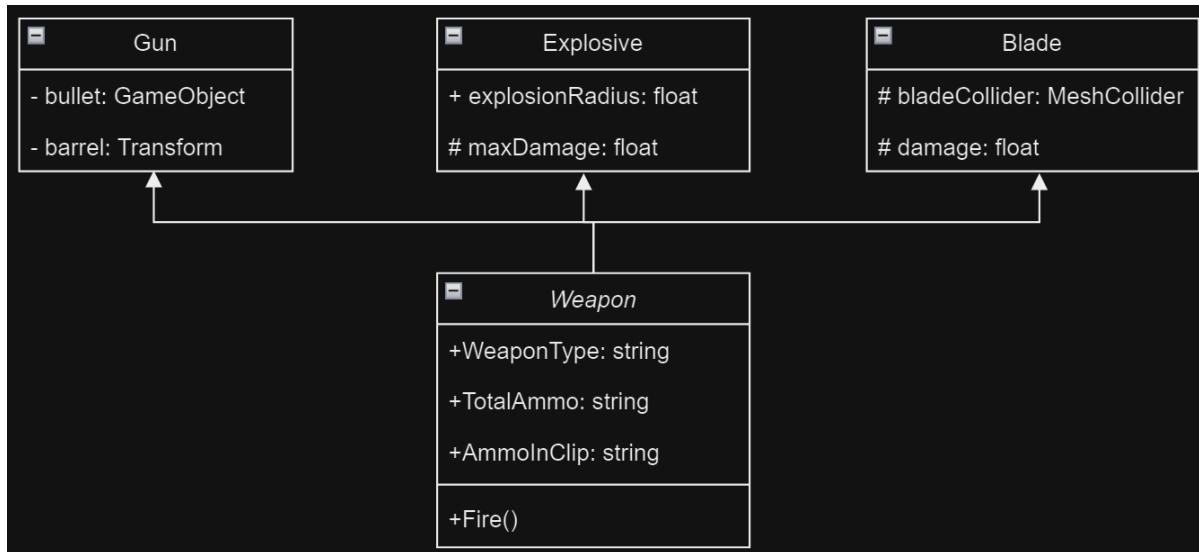


The Entity class system primarily manages the health and damage of the entity it is attached to. However, *Entity* is an abstract class used only as a framework for the *PlayerEntity* and *EnemyEntity*. The only extra functionality the player needs is to upgrade their maximum health, while *EnemyEntity* adds far more functionality to the base class. For enemies, it also provides the functions for its AI and communication systems. While

called `EnemyEntity`, it will also be used for friendly entities, since they have the same use and intent, with only the range of acceptable targets being changed, such as from the Empire to Varsia (the two countries involved in the war).

The main movement the AI will use will be a PID controller, which is used in industrial applications to bring a robot system back to the target without getting stuck in a cycle. It will also be looking for entities opposing it within a cone, limited by its maximum detection range. This tried and tested method is perfect for the enemies.

## Weapons



Weapons are how the damage will be dealt to both players and entities in an interactable form. `Weapon` itself is an abstract class that makes sure all following weapons have the correct fields that the save system uses to save their data, while also making sure that weapons have the same public `Fire()` method that will be used allow scripts that interact with the weapon to use them. This system is also useful for when SillyAustralian wishes to add in more weapons, as OOP makes it easy to add in more weapons, while retaining the previous features, and also combining different weapons for more effects.

## Iterative Tests

These tests will be used constantly during development and will also be updated for whenever new bugs are found and hence test data needs to be created for it. Each test will check for valid, boundary, and erroneous data where applicable. Valid data must pass no matter what, and similarly should boundary data. Meanwhile, erroneous data should not pass through and this may be data that is redundant and should not affect the expected output. All of the 3 tests are important as a game is likely to be pushed to

its extremes in all shapes and forms during gameplay, and so to keep the player's experience as enjoyable as possible, the number of issues and bugs that slip through must be kept to a minimum.

## Strafing

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the W/A/S/D keys alone and in combinations while moving at a wall	These keys allow the player to move in 8 directions once used in the correct combinations and moving at a wall which is required to test the algorithm	The character will accelerate in the direction held until the maximum speed is reached or the direction is released		
Boundary	Pressing opposing directions – A and D, S and W	This is possible input that a player might momentarily input as they switch directions, but since it is created from both valid inputs, but results in a different output, it should also be tested.	Combining these keys should result in no movement since they lead to movement in opposing directions		
Erroneous	Pressing another key (e.g. M)	Keys not used for movement should not produce any movement	The player will remain still		

## Looking

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Moving the mouse up and down	This is the entire input base that the player should be using for looking around	The character will look around normally		
Boundary	Moving the mouse really far	This is to check that there is no snapping in the movement and that the system is capable of large movement values	The character will look around really fast		
Erroneous	No movement	It is important to check that the system does not continue drifting while the player is not moving the mouse	The camera will be still		

## Boosting

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the boost key	This should be the only key used for boosting, and hence should respond appropriately	The player's speed increases by a huge amount, and then the speed cap is set		
Erroneous	Pressing another key (e.g. M)	Keys not used for boosting should not	The player will remain at their current		

		activate it	speed		
--	--	-------------	-------	--	--

## Jumping

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the jump key on the ground	As the only input being used to jump, it is important that it works, and that it has not been affected by the double jump	The character will move upwards, then move back down, and no air jumps are used		
Valid	Pressing the jump key in the air while moving upwards	This tests the player's ability to double jump	The player will move upwards faster, using an air jump, and will fall back down eventually		
Valid	The player pressing the jump key while in mid-air when they have no air jumps left	Since the player is likely to have used all their double jumps, they should no longer be able to jump in mid air	The player will continue accelerating downwards		
Boundary	Touching a surface when the player has no air jumps left	The player will usually exhaust all their double jumps, and they need to be able to get them back	airJumpsLeft will be set to airJumpsTotal		
Boundary	Pressing the jump key in the air while moving downwards	The player should still be able to get the full use out of their double jump without	The player will move upwards, using an air jump, then move down		

		it being a waste			
Erroneous	Pressing another key (e.g. M)	Since there is only one input, nothing else should cause the player to jump, and if there is, that means something is significantly broken	The player will remain still		

## Crouching

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the crouch key at low speeds	This is input that the player will use, and the state needed to crouch, and hence needs to be tested appropriately	The player's height will be shrunk		
Valid	Pressing the crouch key at high speeds	This is input that the player will use, and the state needed to slide, and hence needs to be tested appropriately	The player's height will be shrunk, and they will gain a small bit of speed, before slowing down		
Boundary	Pressing the crouch key at high speeds and then again quickly	The player may use this to chain together a lot of speed and therefore should be tested corrected	The player's height will be shrunk, and they will gain speed before returning to the standing height.		
Erroneous	Pressing	An input not	The player		

	another key (e.g. M)	related to the test should not affect the test	will remain at their standing height		
--	-------------------------	--	--------------------------------------	--	--

### Different crouch keys

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the toggle crouch key	Each input must be tested separately to verify that they work	The player will crouch		
Valid	Holding the hold crouch key	Each input must be tested separately to verify that they work	The player will crouch for the duration that the key is held		
Boundary	Pressing the toggle crouch key and then the hold crouch key, and then releasing it	As was specified in the first interview, the held crouch must take priority over	The player will crouch and then uncrouch when the hold crouch key is released		
Erroneous	Pressing another key (e.g. M)	An input not related to the test should not affect the test, and hence checks that the input system is still working as expected	The player will remain at the same height		

### Looking and strafing combined tests

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Moving the mouse up	This combines all	The character will		

	and down while moving in a circle	the looking actions with all the movement options and is also covers the most typical of a player's movement style	look around and the player will move around relative the current looking direction.		
Boundary	Moving the mouse really far while moving in a circle	This is to check that there is no snapping in the movement and that the system is capable of large movement values, while maintaining movement	The character will look around really fast and will move accordingly		
Erroneous	No movement	It is important to check that the system does not continue drifting while the player is not moving the mouse	The camera will be still, and the player will not move		

## Fire

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the fire button while there is enough ammunition remaining	This is the most common action the weapon will encounter and therefore is suitable as a valid test	The gun will fire 1 bullet		

Boundary	Pressing the fire button while there is no ammunition remaining	It is important the system still acts accordingly when there is no ammunition in the clip remaining	The gun will reload		
Erroneous	Pressing another key (e.g. M)	Any inputs not related to the system should not affect it	The gun will not fire		

## Reload

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the reload button when there is no ammunition left in the clip, and there is enough ammunition remaining in total	This is the next most common action the weapon will encounter and therefore is suitable as a valid test	The gun will reload back to full		
Boundary	Pressing the reload button when there is ammunition left in the clip	It is important the system still acts accordingly when the player reloads before they need to	The gun will reload back to full and only take as much ammo as needed		
Erroneous	Pressing another key (e.g. M)	Any inputs not related to the system should not affect it	The gun will not reload		

### Swap weapon

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Scrolling the scroll wheel in either direction	The player will want to change the current weapon based on the scenario and therefore this is suitable as a valid test	The selected weapon will change relative to the direction the player scrolled in		
Boundary	Scrolling very fast in either direction	It is important the system still acts accordingly when the player may put it under stress	The currently selected weapon should change accordingly		
Boundary	Scrolling when all the weapon slots are not filled	It can occur that all the player's weapons slots are not filled and as such the scrolling should still respond correctly, even though this is a rare case.	The system should continue wrapping with the remaining weapons and change the weapons accordingly.		
Erroneous	Pressing another key (e.g. M)	Any inputs not related to the system should not affect it	The current weapon will remain selected		

### Add weapon

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Adding a weapon	The player will want to	The weapon will be added		

	while there is space available	change the current weapon based on the scenario and therefore this is suitable as a valid test	to the list with no other weapon removed, and will swap to that weapon		
Valid	Adding a weapon while there is no space available	The player will want to change the current weapon based on the scenario, and this is the more common way to acquire a weapon	The weapon will replace the current weapon		
Erroneous	Pressing another key (e.g. M)	Any inputs not related to the system should not affect it	The current weapon will remain selected		

### Weapon pointing to target

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Looking from a close object to a far object or vice versa	While purely cosmetic, it will provide helpful information to the player on where they are pointing	The weapon will point closer to the body of the player and then further away, pointing at the object at the centre of the screen		
Valid	Not moving the mouse after moving it	The function should finish its last action	The weapon will finish its rotation.		
Valid	Not moving the mouse	Since the input data is	The weapon will not rotate		

		the same, the output should also be the same			
Boundary	Looking from a close object to a far object or vice versa while the player is moving quickly	The function should work independent of the player's movement, and the physics engine of Unity can get a dodgy at high speeds	The weapon will point closer to the body of the player and then further away, pointing at the object at the centre of the screen		
Erroneous	Pressing another key (e.g. M)	Any inputs not related to the system should not affect it	The weapon will not rotate		

## Spike Testing

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Running into the spike	This is a basic test for damage as well as well as verifying environmental damage and whether the player's health is updated on the HUD	The player will take damage as is detailed, and this will be shown on the HUD	Passed	
Boundary	Running into a spike with 1 HP (or enough for the next hit to kill the player)	This will test whether the kill function works	The death screen will show up	Passed	

## Entity Movement

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	The target is outside NearRange	As the most common distance for the target from the player it should work as the valid test	The entity will approach the target and then settle at some point between NearRange and FarRange		
Boundary	The target is inside NearRange	Being very close to the target can cause the system to apply too much force and cause the entity to fly off.	The entity should move away and then return to an appropriate distance quickly		
Boundary	The entity is at the target	The entity reaching the target is a required part of its movement while it is in the Searching state, and as such will need to be tested	The entity will not move		
Erroneous	No target is set	This is a test for a potential error in case the target that the entity is tracking disappears and hence should be handled appropriately	The entity will not move		
Erroneous	The target is out of reach	It is incredibly likely that the	The entity will attempt to		

	of the entity	player will get somewhere that the entity will be unable to reach, and as such it should be able to handle with this	reach as close to the target as possible		
--	---------------	--	--	--	--

## Entity Searching

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	The player is in front of the entity	This is the most typical way that the player will be detected by the entity will detect the player	The entity will enter combat mode		
Valid	The player is not in front of entity	The drone will not always see the player and as such must search as usual	The entity will continue searching and moving towards the target		
Boundary	The player is on the edge of the entity's view	The player might be moving quickly and only be in the drone's view for a moment and as such this must be tested appropriately	The entity will enter combat mode		
Boundary	The player is on the edge of the entity's detection range	The player may attempt to escape the drone by being on the edge of its view, and hence it	The entity will enter combat mode		

		should still detect the player			
--	--	--------------------------------	--	--	--

## Entity Combat

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	The player is in front of the entity while it is in combat mode	This is the most typical way that the player will be detected by the entity will detect the player	The entity will fire its weapon and move away		
Valid	The player is not in front of entity while it is in combat mode	The drone will not always see the player and as such must rotate towards them	The entity will rotate towards the player		
Boundary	The player is on the edge of the entity's view while it is in combat mode	The entity also needs to be able to conserve ammunition and this makes sure that they do not fire at all moments	The entity will rotate towards the player		
Boundary	The player is on the edge of the entity's detection range while it is in combat mode	The player may attempt to escape the drone by being on the edge of its view, and hence it should still detect the player	The entity will chase after the player and fire if possible		

## Saving and Loading

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Saving a new save with others already being made	The most typical action the player will be taking is this, and therefore it should be tested properly	A new save will be created with the newest number		
Valid	Loading a save file	It goes hand in hand with saving, and will also be a core function of the system	The save file is opened and the data is returned		
Boundary	Saving a file when none of the directories or files exist	This is necessary to test for since this will be the state of the user's system when they first launch the game	All the appropriate files and folders are created		
Error	Passing a file to load that doesn't exist	In case the lookup ends up corrupted, or the user deletes certain files, the function should not crash, and instead log the error	Error code is logged		
Error	'SaveLookup.txt' is not created	This will occur when the player opens the game for the first time, and must be	The file is created		

		handled properly			
--	--	------------------	--	--	--

### Save Data Application

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Normal SaveData	This is the standard action that will call	A SaveSlot list is returned with the correct information		
Error	The weapon ID is incorrect	This may occur when the game developer puts in the incorrect ID for the prefab in the WeaponType property in the Weapon script	An error is logged, and the weapon is not loaded in.		

### Displaying Saves Normally

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	The 'Saves' folder filled with slots	The player will have at least 1 save slot on a typical opening of the game, and this should be displayed properly	The right number of saves are seen in the scroll view		
Boundary	The 'Saves' folder is empty	This can occur if the player opens the game, but then closes it before creating a save	No saves are seen		

## Displaying Saves when making a new save slot

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Making a new save slot when the folder (Saves) has some already	The player will have at least 1 save slot on a typical opening of the game, and this should be displayed properly	The number of save slots increases by one, and is displayed in the scroll view		
Boundary	Making a new save slot when the folder (Saves) is empty	This will be the state of the player's game on the first opening of the game	The new save slot is shown as the only one		

## Selecting a new menu

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Selecting a new menu	This is the standard operation this system will experience and hence is suitable for a valid test	The current menu will be disabled and not interactable		
Valid	Going back	This will make sure that the lastMenu variable is working appropriately	The previous menu will be reenabled and the current one will be disable		
Erroneous	Requesting a menu that isn't in the dictionary	In case the developer puts in a wrong system, the system should not	A debug message is logged with the error and the current menu stays		

		crash	up		
--	--	-------	----	--	--

## Player Death

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	The player dying	The player is expected to die when playing the game, and hence must be tested correctly	The player's movement stops, and they can no longer control the character, and the death screen is shown		
Valid	Clicking to reload the game	The player will need to reload their game after they die	The previous save is loaded just like a level load, and the player regains control		

## Quitting

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	The button is pressed when the game is in the editor	The player will eventually want to close the game, and as such this ease-of-use method will be required.	The editor's play state is set to false, and the preview stops		
Valid	The button is pressed when the game is an executable	The player will eventually want to close the game, and as such this ease-of-use method will be required.	The game will close and leave the desktop		

## Loading Screen

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Selecting to load a save	This will be the easiest way to test the loading screen and makes a suitable test since this tests the function that runs the screen	The current menu will be disabled, and the loading screen will come up. When the loading is completed, it will show the button with a full progress bar		
Erroneous	Loading a scene that doesn't exist	In case the save file is wrong, the system should not crash	A debug message is logged with the error and the current menu stays up		

## Moveability and Cursor state when loading a scene

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Loading a new scene which the player can't move in	This will be required most surely when the Main Menu is loaded in and will be the one thing every player will interact with.	The player will be unable to move, the cursor will be unlocked, the player will be in the right location, and the right menu is displayed		
Valid	Loading a scene which the player can move in	The player will need to load into a new level as they play the	The player will be able to move, the cursor will be locked, the		

		game, or when they load a save, and hence this must be tested properly	player will be in the right location, and the right menu is displayed		
--	--	--	---	--	--

## UI Audio

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Hovering over the button	The player will be hovering over buttons constantly as they decide what to click	The UIHover sound effect is played		
Valid	Not hovering the button	On the other hand, the player may also skip certain buttons, especially when the buttons do not take up the majority of the scene.	No sound effect will be played		
Valid	Clicking on the button	The player will always have to click a button to progress through the menus	The UISelect sound effect is played		
Boundary	Hovering over and then off the button	Most of the time, the player will hover over a button but not click it, and as such this needs to be tested correctly	The UIHover sound effect is played once as the mouse hovers over, but not off		

## Volume Slider

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Not changing the slider	Most of the time, the player will not need to change the volume of the game when they open the menu, and as such this needs to be tested	The volume remains at its current value		
Valid	Changing the slider to any value	Typically, the player will need to set the volume to an intermediary volume for their needs	The volume will be adjusted according to the slider		
Boundary	Moving the slider to the minimum or the maximum	The player may need to mute the game, or return it to its original value, and this should be tested	The game's volume will be muted (if the slider is set to the minimum) or return to the normal value (if the slider is dragged to the maximum)		
Error	Dragging the slider out of its limits	The player will not attempt to drag the slider carefully, and will typically drag it past its limits to set the minimum and maximum,	The slider will stay at either the minimum or maximum		

		and this should be tested appropriately			
--	--	---	--	--	--

## FOV Sliders

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Not changing either slider	Most of the time, the player will not need to change their FOV for either option when they open the menu, and as such this needs to be tested	The FOV and ADS FOV remains at its current value		
Valid	Changing the FOV to any value	Typically, the player will need to set the FOV to an intermediary value for their needs	The FOV slider will be set to the correct value		
Valid	Changing the ADS FOV to any value	Typically, the player will need to set the ADS FOV to an intermediary value for their needs	The ADS FOV slider will be set to the correct value		
Boundary	Moving the either slider to the minimum or the maximum	While uncommon, some players may use extreme FOVs, and this should be tested	The appropriate slider will be set to either the minimum or the maximum		
Error	Dragging either slider out of its	The player will not attempt to drag either	The slider will stay at either the minimum		

	limits	slider carefully, and will typically drag it past its limits to set the minimum and maximum, and this should be tested correctly	or maximum		
--	--------	--	------------	--	--

## Wallrunning

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing forward on the wall	This is the most likely action that the player is going to do when they are on the wall, and as such, this should be the highest priority test	The player will move forward on the wall		
Valid	Facing perpendicular to the wall	The player may want to dismount from the wall	The player will gain a speed boost as they disconnect from the wall and enter a walking state.		
Valid	The player pressing the jump key while on the wall	Since the aim of the game is to maintain speed, the player will want to jump away from the wall instead of the other methods to	The player will dismount from the wall and gain some speed in the direction of the wall normal and		

		maximise the speed gain	upwards		
Boundary	Slowing down on the wall below a threshold	The player may attempt to turn around on the wall, or slow down, and it makes sense for this to mean that they can no longer hold onto the wall.	The player is pushed away from the wall		
Boundary	Pressing the jump key in the air while moving downwards on the wall	The player is still affected by gravity, if only by a little bit, so the player can end up moving downwards, which may make the jump useless. Hence this needs to be counteracted	The player will dismount from the wall and gain some speed in the direction of the wall normal and upwards		
Erroneous	Pressing another key (e.g. M)	An input not related to the system should not affect it	The player will slow down on the wall due to friction		

### Transpiling a line

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Reading normal syntax	The majority of the transpiler's job is to deal with the syntax, and this makes sense as a valid test	The syntax will be replaced with the appropriate value		

Valid	Reading a backslash	The transpiler must also be able to deal with an escaped character	The following character is skipped over		
Boundary	Reading a character that is not part of the syntax	The majority of the line will be like this, but the transpiler should not interpret it as syntax	The character is passed through the system		

### Dialogue Interaction

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Requesting a line	The interaction will always attempt read a normal line, and so this must be tested correctly	The line is put onto the subtitle window, which is displayed, and then fades away after 2 seconds		
Valid	Reading a line made of multiple pieces	Certain pieces of text may need to be repeated, and this can be made easier by requesting multiple lines from the database	The lines are pieced together and put onto the subtitle window, which is displayed, and then fades away after 2 seconds		
Boundary	Reading a second line after reading 1	The interaction will rarely end with one click, and this needs to be	The second line is added below the first in the subtitle menu, and		

		handled properly	fades away after 2 seconds		
Error	Requesting a line or actor that doesn't exist in the database	The developer may forget to fill in the database appropriately, and this should be handled	An error is raised		
Error	Reading a line that is lacking information	The developer may forget to complete the .dlg file, this must not be passed to the rest of the tokenizer	An error is raised		
Error	Reading an unrecognised token	If the developer has not written the .dlg file properly, the tokenizer should warn them	An error is raised, specifying information on the error		

## Post Tests

While containing similar test data to the iterative tests, these tests will be run at the end of the timeframe and will be used to evaluate how the project has progressed and where it has succeeded and where it has failed. This section will be updated as more features are added. Some of these will be rated using the Likert Scale, which rates how well one agrees to the statement supplied. Some tests may not be listed here as they are the same, or contain minor edits, to the iterative tests above

## Robustness

Type	What data fits criteria	Justification	Expected	Actual
Normal	Play the game in the normal scene, with few	The player will typically experience the game like this, with a reasonable number of objects	The game not run below 30 fps	

	elements			
Intense	Playing the game in a scene with lots of entities	In case the player is chased by many enemies, the game should be able to handle this well	The game not run below 30 fps	

## Glitches

Type	Justification	Example	Severity	Steps to reproduce
Input	Any glitches with the player's input will be unsatisfactory, and may frustrate the player	An input is missed		

## Strafing

What data fits criteria	Justification	Expected	Pass/Partial/Fail	Comments
Pressing the left input, then releasing	These 8 tests relate to the different valid input directions that the player can supply, and every single one should work	The player moves left and stops		
Pressing the right input, then releasing		The player moves right and stops		
Pressing the up input, then releasing		The player moves up and stops		
Pressing the down input, then releasing		The player moves down and stops		
Pressing the left and up input, then releasing		The player moves north-west and stops		
Pressing the left and down input, then releasing		The player moves south-west and stops		
Pressing the right and up input, then releasing		The player moves north-east and stops		
Pressing the right and down input, then releasing		The player moves south-east and stops		

Pressing the right and left input, then releasing	While not leading to any overall movement, it is an input the player can make that is made of completely valid inputs	The player doesn't move		
Pressing the up and down input, then releasing	Testing for the lack of inputs is also something that can happen on any frame, and must be tested correctly	The player doesn't move		
Pressing no input	This is invalid and should not affect the strafing system, and must be dealt properly	The player doesn't move		
Pressing an unrelated key (e.g. M)		The player doesn't move		

## Jumping

What data fits criteria	Justification	Expected	Pass/Partial/Fail	Comments
Pressing the jump input	This is the key the player will press when they want to jump up, and so should be tested correctly	The player moves upwards		
Pressing the jump input while in midair and moving upwards with air jumps remaining	This will test the player's ability to double jump, which will allow the player to reach slightly further distances	The player moves upwards		
Pressing the jump input while in midair and moving downwards		The player moves upwards		
Pressing the jump input		The player continues		

while in midair and moving upwards with no air jumps remaining		moving upwards that they are moving – the input is effectively ignored		
Pressing the jump input while in midair and moving downwards with no air jumps remaining		The player continues moving downwards that they are moving – the input is effectively ignored		
Pressing no input	Testing for the lack of inputs is also something that can happen on any frame, and must be tested correctly	The player doesn't move		
Pressing an unrelated key (e.g. M)	This is invalid and should not affect the jumping system, and must be dealt properly	The player doesn't move		

## Crouching

What data fits criteria	Justification	Expected	Pass/Partial/Fail	Comments
Pressing the toggle crouch, then releasing	This is one of the keys that the player can press to crouch	The player will shrink in height		
Pressing the hold crouch, then releasing	This is other key that the player can press to crouch, and is likely to be the more commonly used one	The player will shrink in height, and then return to the standing height		
Pressing the toggle crouch, then releasing.	The player may use both inputs to crouch, and	The player will shrink in height, and then return		

Then pressing the hold crouch, and then releasing it	this should not lead to any conflicts	to the standing height when the hold crouch is released		
Pressing no input	Testing for the lack of inputs is also something that can happen on any frame, and must be tested correctly	The player remains at the standing height		
Pressing an unrelated key (e.g. M)	This is invalid and should not affect the crouching system, and must be dealt properly	The player remains at the standing height		

## Using weapons

What data fits criteria	Justification	Expected	Pass/Partial/Fail	Comments
Pressing the fire input with an equipped weapon	This is the key the player will press when they want to fire the weapon, and so should be tested correctly	The equipped weapon will fire a bullet		
Pressing the fire input with no equipped weapon	The player may accidentally press this input, and as such this needs to be handled	Nothing will happen to the weapons		
Pressing the swap weapon input with no weapons while facing a weapon	The player will need to do this at the start of the level, and so nothing should break when they do so	The weapon will become equipped and move to the player's hand and be added to the list of weapons		
Pressing the	The player may	The weapon will		

swap weapon input with some weapons while facing a weapon	do this when they want to pick up more weapons	become equipped and move to the player's hand and be added to the list of weapons		
Pressing the swap weapon input with as many weapons as they can hold while facing a weapon	The player may do this when they run out of ammo on their current weapon	The weapon will become equipped and replace the previously equipped weapon		
Pressing the swap weapon input while not facing a weapon	The player may accidentally press this input, and as such this needs to be handled	Nothing happens to the weapons		
Pressing the reload input with an equipped weapon	The player may want to reload their weapons early while they have cover and some breathing time	The currently equipped weapon reloads as much ammunition as it can		
Pressing the reload input with an equipped weapon, but no ammunition remaining	The player may want to reload their weapons earlier, but they shouldn't be able to in this case	Nothing happens to the weapons		
Pressing the reload input with no equipped weapon	The player may accidentally press this input, and as such this needs to be handled	Nothing happens to the weapons		
Pressing the zoom input with an equipped weapon, then releasing it	The player will need to zoom in when they want to have a precise shot, and so this should be tested	The player's FOV zooms in, then back out.		

	properly			
Pressing the zoom input with no equipped weapon	The player may accidentally press this input, and as such this needs to be handled	Nothing happens to the weapons		
Pressing no input	Testing for the lack of inputs is also something that can happen on any frame, and must be tested correctly	Nothing happens to the weapons		
Pressing an unrelated key (e.g. M)	This is invalid and should not affect the jumping system, and must be dealt properly	Nothing happens to the weapons		

### Taking damage

What data fits criteria	Justification	Expected	Pass/Partial/Fail	Comments
Get hit	This should lower the player's health, but not enough to kill them, so it should still be visible to the player	The player losses health as shown on the HUD		
Getting hit enough to lose all of their health	This will cause the player to die, and tests the death screen as well	The player is unable to control their character, and the death screen shows up		
Pressing 'Load' on the death screen	The player will most definitely want to do this when they die, and should be tested properly	The game will reload the player's save data and the player will be able to play		

		once again		
--	--	------------	--	--

## Saving

What data fits criteria	Justification	Expected	Pass/Partial/Fail	Comments
Pressing any save button	The player will want to save their current progress and not lose it, and this should work well	A new save file is created on the player's computer		
Making a new save slot	The player may want to replay the game at some point completely fresh, and this should allow them to make completely fresh experience	A new save slot folder is created on the player's computer, along with a new basic save		
Pressing any load button	The player will most definitely want to load their save when they get stuck or in other situations	The game will reload the player's save data and the player will be able to play once again		
Selecting to load a slot in the slot menu	The player will want to load into a save slot when they open the game and so this must be handled properly	The game will load the save with a loading screen, and load into the correct level		

## Dialogue

What data fits criteria	Justification	Expected	Pass/Partial/Fail	Comments
Pressing the interact key when facing a dialogue object	This is the key the player will press when they want to interact with the	A new dialogue line is displayed in the subtitle box that shows up		

	dialogue, and so should be tested correctly			
Pressing the interact key when not facing a dialogue object	The system should consider this an invalid input since the input should mean nothing when not facing an interactable	Nothing happens		
Pressing the interact key when the next line is an if statement that passes	This will test whether the system can handle reading flags	The variable is read, and the line number jumps to the section		
Pressing the interact key when the next line is an if statement that fails	This will test whether the system can handle reading flags	Only the line number increments		
Pressing the interact key when the next line is an assignment statement	This will test whether the system can handle setting flags	The variable is set in the save data flags		
Pressing the interact key when the next line is a choice statement	This will test whether the system can handle choices properly, and respond appropriately	The choice menu shows up with each choice available to the player in a grid pattern		
Pressing no input	Testing for the lack of inputs is also something that can happen on any frame, and must be tested correctly	The player doesn't move		
Pressing an unrelated key (e.g. M)	This is invalid and should not affect the dialogue system,	Nothing happens		

	and must be dealt properly			
--	----------------------------	--	--	--

## Entity behaviour

What data fits criteria	Justification	Likert Scale	Comments
The enemy successfully searches and finds me	The enemies will need to search for the player, which will make the experience better for the player, since they can attempt to avoid the sight of the enemy		
The enemy successfully targets and chases me	The enemy needs to be able to turn towards and chase after the player to be a potential threat to them.		
Once targeted, the enemy successfully fires at me.	For the enemy to be a threat to the player, and encourage the intended behaviour, it needs to fight against the player		
The friendly entities follow the player successfully	Friendly entities are useful to the player only when they are able to reach the same destination as the player, and this requires that they follow the player		
The neutral entities run away successfully	Entities that cannot fight should constantly attempt to run from any opposition, signalling to the player that they are not to be fought		
The friendly entities help the player when fighting	The purpose of the friendly entities is to fight alongside the player, which needs		

	them to fire against the enemies		
--	----------------------------------	--	--

## Usability Tests

These tests will test how easy it is to navigate the menus and see elements on them.

Statement	Justification	Likert Scale	Comments
The main menu is easy to navigate	This is the first screen that player will see when they open the game, and so it should be as easy to use as possible		
The slot menu is easy to navigate	This is one of the more important menus that the player may interact with and as such it should not be difficult to navigate		
The pause menu is easy to navigate	The player will have the ability to do quite a lot during the game while accessing this menu, so anything that makes this menu difficult could lead to a frustrating experience		
The death menu is easy to navigate	The player is likely to visit this menu quite often, and so it should be as simple as possible to allow them to leave it as fast as possible		
The options menu is easy to navigate	The player will have a lot of control within this menu and so this menu should be usable and simple to use		
I can see my health on the HUD clearly.	The player should be able to see their health easily since this is an important part of the game and		

	the player may be in danger if they can't see it		
The health does not obstruct my view of the game	Conversely, if the health is too large, then the player will not be able to see the game's world and may be in more danger		
I can see my current weapon on the HUD	The player should be able to see what weapon they have equipped and any information about that weapon		
I can see the previous and next weapons on the HUD	The player should also be able to see what weapons they may be changing too as this also important information		

## Development

### Prototype 1

#### Input Detection

To handle the detection of many different actions from many different input sources, I will be using Unity's built-in Input System package. Within this, I can set up different input maps, such as 'Player', and each of these maps have different actions, such as 'Fire', and 'Jump'. I can easily assign to each of these actions multiple different input types that activate them. Actions are a special version of an event, which is a way for functions to be run when something happens. So, when the user presses one the inputs, the related action emits a signal to all the event's subscribers (the term for functions listening for the event), which all activate simultaneously. This is how the game senses the inputs.

Since multiple input maps can have actions that are enabled with the same inputs, only one input map must be enabled at any time, and the same instance of the input map must be used across all scripts. To follow this criterion, a script called PlayerManager.cs exists, of which one of its functions is to instantiate a new set of inputs and enable the Player input map the moment the game starts. The reference to this map is public, meaning any class wishing to get the inputs of the user needs only reference this

variable. Similarly, if the map needs to be changed, such as one used to handle UI, any script can do so, and it will permeate across all, making sure that no unintended actions occur when the player is intending to do something else.

### Testing input maps

## Movement

Movement is the most important aspect, and as such, a lot of time has been spent implementing many aspects of the movement. I originally started with a rigidbody method. This works in tandem with the built-in Physics system of Unity where all movement inputs are applied with forces to the player. First, we define many variables that are required, such as the maximum walking speed, the friction, and the current movement state, among others.

```
[Header("Movement")]
[SerializeField] private PlayerMovementState movementState;
[SerializeField] private float friction = 0.8f;
[SerializeField] private Vector3 velocity;
[SerializeField] private Vector2 horizontalVelocity;
[SerializeField] private Vector2 inputDirection;
[SerializeField] private float movementAcceleration = 2f;
[SerializeField] private float maxWalkSpeed = 5f; // Usain Bolt speeds, average
human pace is a quarter of this

private Rigidbody player;
private PlayerInputs playerInputs; // Use if using the C# Class
// private PlayerInput playerInput; // Use if using the Unity Interface

enum PlayerMovementState
{
    idle,
    walking
}
```

When the object is initialised, we need to fetch the player's rigidbody, along with enabling the player's inputs. Also, at the start of the game, since the player is not moving, they are idle and so this must be set as well.

```
private void Start() {
    player = GetComponent<Rigidbody>();
    playerInputs = new PlayerInputs();
    playerInputs.Player.Enable(); // Enabling only the Player input map
    movementState = PlayerMovementState.idle;
}
```

After that, we begin the code for what happens every frame (in Update() ). This includes retrieving the player's velocity and copying it to an internal variable, along with deciding

what the player's movement state is. Then we need to take the movement inputs, decide the required movement amount, then apply friction. Furthermore, we need to make sure to limit the player's speed, and then finally send the resulting change to the Physics system.

```
private void Update() {
    velocity = player.velocity;
    horizontalVelocity = new Vector2(player.velocity.x, player.velocity.z);
    if (Mathf.Abs(horizontalVelocity.magnitude) < maxWalkSpeed){
        movementState = PlayerMovementState.walking;
    }
    if (velocity == Vector3.zero){
        movementState = PlayerMovementState.idle;
    }
    Movement();
    horizontalVelocity *= FrictionMultiplier();
    horizontalVelocity = ClampSpeed();
    velocity = new Vector3(horizontalVelocity.x, velocity.y, horizontalVelocity.y);
    player.AddForce(velocity - player.velocity, ForceMode.VelocityChange);
}
```

You have to make sure to take the absolute of the ground velocity's magnitude (horizontalVelocity.magnitude) when comparing it to the maxWalkSpeed. FrictionMultiplier() is a relatively simple function, as it is only an if statement. Multiplying the output to a vector will result in the remaining velocity if friction was applied.

```
private float FrictionMultiplier(){
    if (inputDirection.magnitude == 0)
        return 1 - friction;
    } else {
        return 1;
    }
}
```

Similarly, ClampSpeed() is also simple, this time being a switch case statement, since all it checks against is the enumerator I made earlier.

```
private Vector2 ClampSpeed(){
    switch (movementState){
        case PlayerMovementState.walking:
            return Vector2.ClampMagnitude(horizontalVelocity, maxWalkSpeed);
        default:
            return horizontalVelocity;
    }
}
```

It uses the built-in function Vector2.ClampMagnitude, which scales the first argument (a Vector2) such that if it is higher than the second argument, it makes the vector smaller until it reaches the second argument. In case the state reached is not recognised, by default the vector is returned to not mess with it.

Finally, Movement() is the most complicated function out of all of these. So long as the player's wants to move, it will calculate how fast they should be accelerated, and the resulting displacement changed in the time of the last frame. However, it may be that the player wants to move in a direction that they are currently not moving in, and as such these 2 directions need to be smoothly interpolated.

```
private void Movement() {
    inputDirection = playerInputs.Player.Movement.ReadValue<Vector2>().normalized;
    if (inputDirection.magnitude != 0) {
        float speedMultiplier = CalculateAccelerationMultiplier();
        Vector3 multipliedVelocity = new Vector2(inputDirection.x, inputDirection.y)
            * movementAcceleration * speedMultiplier * Time.deltaTime;
        Vector3 directedVelocity = new Vector2(inputDirection.x, inputDirection.y) *
            movementAcceleration * 20 * Time.deltaTime;
        float alignment = Vector3.Dot(player.velocity.normalized, directedVelocity.
            normalized);
        Vector3 lerpedVelocity = Vector3.Slerp(directedVelocity, multipliedVelocity,
            alignment);
        horizontalVelocity += new Vector2(lerpedVelocity.x, lerpedVelocity.y);
    }
}
```

The function internally calls CalculateAccelerationMultiplier() which also uses the enumerator in a switch case statement to return an appropriate acceleration. This also calls a function that is an equation for how much speed should be gained.

```
private float CalculateAccelerationMultiplier(){
    switch (movementState){
        case PlayerMovementState.walking:
            float clampedMagnitude = Vector2.ClampMagnitude(horizontalVelocity, maxWalkSpeed).magnitude;
            Debug.Log(clampedMagnitude);
            return SpeedFunction(clampedMagnitude, 0.5f, 20);
        default:
            return 1;
    }
}
private float SpeedFunction(float speed, float a, float b){
    return -(Mathf.Pow(speed/a, 4)/(b*b*b))+b;
}
```

This works completely fine, and there were no bugs with this setup, and as such I attempted to speed this up. This is because Mathf.Pow is an extremely slow function for my needs, due to there being lots of overhead in case it needs to handle non-integer powers. Since I only need to do a predetermined, integer amount of exponentiation, I can rewrite my own quick function to do this with multiplication, which is much faster.

```

private float SpeedFunction(float speed, float a, float b){
    return -(Pow4(speed/a, 4)/(b*b*b))+b;
}
private float Pow4(float num){
    return num*num*num*num;
}

```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the W/A/S/D keys alone and in combinations	These keys allow the player to move in 8 directions once used in the correct combinations	The character will accelerate in the direction held until the maximum speed is reached or the direction is released	Passed	
Boundary	Pressing opposing directions – A and D, S and W	This is possible input that a player might momentarily input as they switch directions, but since it is created from both valid inputs, but results in a different output, it should also be tested.	Combining these keys should result in no movement since they lead to movement in opposing directions	Passed	
Erroneous	Pressing another key (e.g. M)	Keys not used for movement should not produce any movement	The player will remain still	Passed	

At this point, I was ready to move onto Jumping, and hence Gravity. Both of these required knowing about if the player was on the floor or not: one cannot jump if they are in the air (note I am implementing only jumping right now, double jump comes later),

and it is faster and a better experience to only apply gravity when the player is not on the ground, otherwise the physics system will have to work overtime trying to make sure you don't fall through the floor. While rigidbodies in Unity do have a built-in toggle to allow them to have gravity or not, it is better to manually implement gravity with my own constant to better control how the player feels in the game.

```
private void Gravity(){
    if (!isGrounded){
        player.AddForce(new Vector3(0, -9.81f, 0), ForceMode.Impulse);
    }
}
```

For this to work, however, we need to find out if the player is on the ground or not. Since finding out if the player is on the ground requires interacting with the Physics system, many of the following built-in functions will be from the Physics library, such as the first typical method of using Physics.CheckSphere(). This queries the physics system on whether there is a collision within a specified sphere, defined by radius and centre.

```
private bool GroundCheck(){
    return Physics.CheckSphere(
        player.transform.position,
        0.5f
    )
}
```

This code seemed to work fine, as when it was run, it did return true when the player was on the ground. To test if it worked when not on the ground, I needed to implement jumping, which should consist of applying a force upwards on the player if they are grounded. With this, we define a new float called jumpForce which we can control in the inspector of Unity.

```
private void Jump(InputAction.CallbackContext inputType){
    if (isGrounded){
        player.AddForce(Vector3.up * jumpForce, ForceMode.VelocityChange);
    }
}
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the jump key	As the only input being used to jump, it is important that it works	The character will move upwards, then move	Failed: The player continued moving upwards	The function might be sensing the player

			back down	and not coming back down afterwards	all the time instead of ignoring it
Erroneous	Pressing another key (e.g. M)	Since there is only one input, nothing else should cause the player to jump, and if there is, that means something is significantly broken	The player will remain still	Passed	

Upon binding this function to the Jump action being performed with `playerInputs.Player.Jump.performed += Jump`, pressing the jump key does indeed cause the player to start rising at a certain speed. Unfortunately, you can still continue jumping infinitely, showing there is indeed a bug with `GroundCheck()`. Upon closer inspection, the query is sensing that it is inside of the player, and hence that is why it is returning true all the time. Consulting the Unity documentation, `CheckSphere()` has an overload that takes an optional layer mask, which will exclude objects on those layers from being sensed.

```
private bool GroundCheck(){
    LayerMask playerLayer = LayerMask.GetMask("Standable");
    return Physics.CheckSphere(
        new Vector3(player.transform.position.x, player.transform.position.y - groundDistance + (0.99f * player.transform.localScale.x * playerCollider.radius) - (player.transform.localScale.y * playerCollider.height)/2, player.transform.position.z),
        player.transform.localScale.x * playerCollider.radius * 0.99f,
        playerLayer);
}
```

While this is fine, grabbing the layer mask every single time `GroundCheck()` is run is extremely inefficient and would hurt the player's performance. Since the value of `playerLayer` never changes either from scene to scene, it is more efficient to grab the reference once at the start of the game when the script is first run, within the `Start()` function.

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the jump key	As the only input being used to jump, it is important that it works	The character will move upwards, then move back down	Passed	

Erroneous	Pressing another key (e.g. M)	Since there is only one input, nothing else should cause the player to jump, and if there is, that means something is significantly broken	The player will remain still	Passed	
-----------	-------------------------------	--	------------------------------	--------	--

The next feature to work on is Boost. This should allow the player to multiplicatively gain speed, so that when they are at higher speeds, they gain a higher boost, encouraging the player to maintain speed. Boost requires that the player's velocity is simply added back onto itself, with the appropriate multiplication. This new state needs to be added to the PlayerMovementState enum, but it need not be added to the acceleration function, since allowing the player to accelerate at such a high speed will make it difficult to manage their speed effectively. This will also need to be added to the if statement for FrictionMultiplier().

```
private void Boost(InputAction.CallbackContext inputType){
    movementState = PlayerMovementState.boosting;
    Vector2 inputDirection = playerInputs.Player.Movement.ReadValue<Vector2>().normalized;
    Vector2 horizontalMovement = new Vector2(player.velocity.x, player.velocity.z).normalized;
    if (inputDirection.x != 0) {
        horizontalMovement.x = Mathf.Abs(horizontalMovement.x) * Mathf.Sign(inputDirection.x);
    }
    if (inputDirection.y != 0) {
        horizontalMovement.y = Mathf.Abs(horizontalMovement.y) * Mathf.Sign(inputDirection.y);
    }
    player.AddForce(new Vector3(horizontalMovement.x, 0, horizontalMovement.y) * boostMultiplier, ForceMode.VelocityChange);
}
```

The player's currently velocity is taken into account and then transformed around to point in the direction that is wanted by the player, so long as that input direction is not 0.

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing Q	This should be the only key used for boosting, and hence should respond appropriately	The player's speed increases by a huge amount	Partially	Friction causes boosted speed to be lost far too quickly, or impossible to get past a certain speed.

Erroneous	Pressing another key (e.g. M)	Keys not used for boosting should not activate it	The player will remain at their current speed	Passed	
-----------	-------------------------------	---	---	--------	--

While boosting, it typically occurs that the player loses the speed that they gained far too quickly, such that it becomes irrelevant to use past a certain speed. While this could be advantageous, the limit is far too low and not enjoyable to utilise boosting at all. While it is still completely usable, for the player's experience, I think it would be better to reduce the amount of friction that the player so that they can maintain the speed far easier and for longer. However, I will leave this decision up to the stakeholders:

**Stakeholders:** Hmm, I do see your point. I think the best solution is indeed your solution, so feel free to change the friction while the player is boosting.

**Me:** If so, how much should I reduce the friction by?

**Stakeholders:** Try out a fifth of the friction and I'll test it out and see how it feels, and then we shall narrow it down from there.

Friction for boost (percentage of normal friction)	Stakeholder comments
20%	This still feels like it is not allowing the player to use the full potential of boost, this needs to go lower.
15%	This feels better than the previous one but it still needs a bit of perfecting. Try going lower.
10%	It feels even better this time! Just to make sure that there isn't a better one, you should make it just a bit lower and I shall see from there.
5%	Yes, the previous one was the best, this is too little friction, and the player doesn't noticeably slow down at all.

Now that I have reached a suitable value for the friction while boosting, I could test it appropriately, since any potential bugs found along the way could potentially be a problem from the specific value of the friction.

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing Q	This should be the only key	The player's speed	Failed: the player went	The game may not be

		used for boosting, and hence should respond appropriately	increases by a huge amount	through a wall	sensing the collision because the player is moving too fast
Erroneous	Pressing another key (e.g. M)	Keys not used for boosting should not activate it	The player will remain at their current speed	Passed	

The player passing through the wall was one of the most simultaneously difficult and simple issues I have encountered so far. I spent far too long trying to find a solution to this problem, consulting Unity forums and Stack Overflow. As such, I consulted my stakeholder on whether to make the player slower, or to rewrite the movement from the ground up using an alternative I found in my research.

#### *Interview*

**Stakeholder:** This is a very drastic solution to the problem. While I absolutely do not want to make the player slower as this is what makes the game fun, this could lose you a lot of time on this project, and as such I will need to cut back on some features.

**Me:** I have already spent a while trying to fix this and without severely limiting and reducing the player's experience, I cannot currently see any other way to get around this. The new method will not be as good as the current one, and it may be more difficult to implement, but it is a fix, nonetheless.

**Stakeholders:** I see. In that case, I'd like to cut back on the weapons and the entities' AI as these are secondary things that we decided on. Feel free to omit implementing the blades and the explosive weapons, and making the entities be on the player side, or even alert a manager. These are all secondary in face of the movement – the primary genre of this game.

Stakeholder Signature:



[-----]

The algorithm that I found is a standard in the industry for dealing with a dynamic object colliding with static objects such as wall. It is called Collide and Slide, where its job is in the name – it collides the object with another and slides it across the other's surface. Certain functions I have already written can be maintained, only needing minor fixes here and there to fit with the controller, that does not use a rigidbody in the same format. Movement() was changed to instead to manually change the velocity off the rigidbody, which is affected by the aforementioned algorithm.

```
private void Movement(){
    Vector2 inputDirection = playerInputs.Player.Movement.ReadValue<Vector2>().normalized;
    if (inputDirection.magnitude == 0) {
        if (isGrounded) {
            player.AddForce(new Vector3(-player.velocity.x, 0, -player.velocity.z) * FrictionMultiplier() * Time.deltaTime, ForceMode.VelocityChange);
        }
    } else {
        float speedDecelerator = CalculateAccelerationMultiplier();
        player.AddForce(new Vector3(inputDirection.x, 0, inputDirection.y) * movementAcceleration * player.mass * Time.deltaTime * speedDecelerator, ForceMode.Impulse);
    }
    if (player.velocity.magnitude < speedCapToDisableOverflow) {
        Vector2 horizontalMovement = new Vector2(player.velocity.x, player.velocity.z);
        float overflowReduction = (Mathf.Abs(horizontalMovement.magnitude) - maxMoveSpeed) * overflowReductionMultiplier;
        if (overflowReduction > 0.1f && overflowReduction < 2f) {
            horizontalMovement.x -= Mathf.Abs(horizontalMovement.x/horizontalMovement.magnitude) * overflowReduction * Mathf.Sign(horizontalMovement.x);
            horizontalMovement.y -= Mathf.Abs(horizontalMovement.y/horizontalMovement.magnitude) * overflowReduction * Mathf.Sign(horizontalMovement.y);
        }
        player.velocity = new Vector3(horizontalMovement.x, player.velocity.y, horizontalMovement.y);
    }
}
```

This time, movement handles the player going over the maxSpeed differently: if the player is below a certain speed, it multiplies the overflow speed by an amount which is subtracted from the current speed to slowly reduce the player's velocity back to the maximum speed. However, if the player is moving above a certain speed cap, the player is deemed as boosting, and the overflow is not reduced. The player also only feels friction once they let go of the movement keys, to make it easier for the player to reach the maximum speed.

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the W/A/S/D keys alone and in combinations	These keys allow the player to move in 8 directions once used in the correct combinations	The character will accelerate in the direction held until the maximum speed is reached or the direction is released	Passed	
Boundary	Pressing opposing directions – A	This is possible input that a player	Combining these keys should result	Passed	Importantly, this also leads to the

	and D, S and W	might momentarily input as they switch directions, but since it is created from both valid inputs, but results in a different output, it should also be tested.	in no movement since they lead to movement in opposing directions		player slowing down, as they should.
Erroneous	Pressing another key (e.g. M)	Keys not used for movement should not produce any movement	The player will remain still	Passed	

To go hand in hand with this, `CollideAndSlide()` corrects the player's velocity as they approach a collider. It predicts where the player will be on the next frame, and if the player collides with something, it will calculate where the player should end up after colliding with that object.

```

private Vector3 CollideAndSlide(Vector3 velocity, Vector3 origin, bool inGravityPass, Vector3 initialVelocity){
    float collisionDistance = (velocity.magnitude + cASMargin);
    RaycastHit hitInfo;
    if (Physics.CapsuleCast(
        point1: player.position + new Vector3(0f, playerCollider.height/4f*playerHeight),
        point2: player.position - new Vector3(0f, playerCollider.height/4f*playerHeight),
        radius: playerCollider.radius,
        direction: velocity.normalized,
        maxDistance: collisionDistance,
        hitInfo: out hitInfo,
        layerMask: playerLayer
    )){
        Vector3 movePlayerToSurface = velocity.normalized * (hitInfo.distance); // - cASMargin
        Vector3 remainder = velocity - movePlayerToSurface;

        if (movePlayerToSurface.magnitude <= cASMargin){
            movePlayerToSurface = Vector3.zero;
        }

        float surfaceAngle = Vector3.Angle(Vector3.up, hitInfo.normal);
        if (surfaceAngle <= maxSlopeAngle){
            if(inGravityPass){
                return movePlayerToSurface;
            }
            remainder = MagnitudeMaintainedProjection(remainder, hitInfo.normal);
        } else {
            float velocityScaler = 1 - Vector3.Dot(
                new Vector3(hitInfo.normal.x, 0f, hitInfo.normal.z).normalized,
                -new Vector3(initialVelocity.x, 0f, initialVelocity.z)
            );

            if(isGrounded && !inGravityPass){
                remainder = MagnitudeMaintainedProjection(
                    new Vector3(remainder.x, 0f, remainder.z).normalized,
                    new Vector3(hitInfo.normal.x, 0f, hitInfo.normal.z)
                ).normalized * velocityScaler;
            } else{
                remainder = velocityScaler * MagnitudeMaintainedProjection(remainder, hitInfo.normal);
            }
        }
        return movePlayerToSurface + CollideAndSlide(remainder, origin + movePlayerToSurface, inGravityPass, initialVelocity, bounceCount + 1);
    }
    return velocity;
}

```

`CollideAndSlide()` is a recursive algorithm, being called on the remaining velocity every time it collides with something. Collisions are detected with a CapsuleCast, which is similar to a RayCast, except it shoots a capsule instead, which is the same shape as the player. If a collision is detected, the player's extra velocity is rotated to be parallel with the collider's normal, of which a percentage is taken away based on how much rotation is required, which is calculated with the dot product. Rotation is done via `MagnitudeMaintainedProjection()`, which simply stores the magnitude of the vector, then uses `Vector3.ProjectOnPlane()` to project it and finally sets the magnitude back to the original. The total of the required velocity to move the player to the surface and the speed for any further movements was returned back, via calling the algorithm again. This recursive nature is stopped when there was no collision.

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the W/A/S/D keys alone and in combinations while moving at a wall	These keys allow the player to move in 8 directions once used in the correct combinations	The character will accelerate in the direction held until the	Failed: Unity ran into maximum recursion depth of 100	Mostly likely comes from there being no maximum bounce count and the

		and moving at a wall which is required to test the algorithm	maximum speed is reached or the direction is released		algorithm just continuously calling itself as it bounces in a corner
Boundary	Pressing opposing directions – A and D, S and W	This is possible input that a player might momentarily input as they switch directions, but since it is created from both valid inputs, but results in a different output, it should also be tested.	Combining these keys should result in no movement since they lead to movement in opposing directions	Passed	Importantly, this also leads to the player slowing down, as they should.
Erroneous	Pressing another key (e.g. M)	Keys not used for movement should not produce any movement	The player will remain still	Passed	

To counteract the infinite recursion, maxBounceCount was added to the function, along with a tracker for how deep the algorithm has gone so far.

```
private Vector3 CollideAndSlide(Vector3 velocity, Vector3 origin, bool inGravityPass, Vector3 initialVelocity, int bounceCount = 0){
    if (bounceCount >= maxBounces){
        return Vector3.zero;
    }
}
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the W/A/S/D keys alone and in combinations while moving at a wall	These keys allow the player to move in 8 directions once used in the correct combinations	The character will accelerate in the direction held until the	Failed: The player slowed down as they approached the wall instead of	Could be caused by the percentage loss being incorrectly applied or there being

		and moving at a wall which is required to test the algorithm	maximum speed is reached or the direction is released	continuing along the path at the correct speed	errors between using normalised and unnormalized vectors
--	--	--	---	--	--

While the error of the stack overflowing was solved, the player now slows down as they approach the wall, which is completely unintended behaviour. However, while scouring the code for what could be causing this, I discovered that rigidbodies had a setting that changed how they detect collisions.

### Collision Detection Discrete

This setting was what I needed to solve the original problem, and which would cause the first code to work correctly when the player was moving at high speeds. This setting has 4 different options:

- Discrete – The least computationally intensive option, but also the least accurate, shown well as it was causing the error the first version of this script was experiencing
- Continuous – This sweeps the object from the location at the start of the frame to the end position and detects if there are any collisions with static objects (e.g. walls) in the sweep, and deals with it appropriately. Does not work with objects that move or rotate
- Continuous Dynamic – Works similarly to continuous except it also queries the other moving entities to see if there is a collision with them as well. Does not work with objects that rotate themselves
- Continuous Speculative – Works differently to the other continuous collision detection methods, thereby making it faster but less accurate. It increases an object's AABB (axis-aligned minimum bounding box) to include the start and end position, then detecting any collisions within that zone. Well known to occasionally detect false collisions and cause havoc. Does work with objects that have rotated themselves.

Out of all these options, the one that I need to use is Continuous Dynamic collision detection, which also allows me to use the original code.

#### Interview

**Me: I've discovered a method for me that would allow me to fix and use the original code, which so far has less errors and issues than the current implementation. Is it fine for me to use that instead?**

Stakeholder: Yes, in that case, go back to the old code, such that we can save time that way. However, is it possible for you to make the original movement feel less slippery while the player is not boosting? I noticed it was not the most responsive due to it, which is hurting the player's experience.

Stakeholder Signature:



[-----]

Now that I have returned to the original code, I need to improve the movement function. Furthermore, this movement does not necessarily allow for multiple movement states such as crouching and sprinting. Due to this, the code was edited to instead use a target which is the player's intended direction and to then work out how much extra speed the player needs to reach that speed. This speed is then projected onto the surface that the player is currently on, with its magnitude maintained. This movement then has a certain acceleration applied, and then friction is applied appropriately. The advantage of doing things like this means that all movement works the exact same way, and the only thing that needs to be changed is the current state's max speed in the switch statement within MaxSpeed(), and the acceleration multiplier which is handled in a similar way with CalculateAccelerationMultiplier(). In this way, the things that change between states are handled externally to the function and hence the function's core can stay the same way with as little if statements as possible to slow the program down.

```

private void Movement() {
    float maxSpeed = MaxSpeed();
    float acceleration = CalculateAccelerationMultiplier();
    Vector3 target = new Vector3(inputDirection.x, 0, inputDirection.y);

    if (surfaceVelocity.magnitude > maxSpeed) {
        acceleration *= surfaceVelocity.magnitude / maxSpeed;
    }
    Vector3 direction = target * maxSpeed - surfaceVelocity;
    float directionMag = direction.magnitude;
    direction = Vector3.ProjectOnPlane(direction, groundNormal).normalized * directionMag;
    Vector3 targetMovement = direction.normalized * acceleration;
    targetMovement -= direction * FrictionMultiplier();

    player.AddForce(targetMovement * Time.deltaTime, ForceMode.VelocityChange);
}

```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the W/A/S/D keys alone and in combinations while moving at a wall	These keys allow the player to move in 8 directions once used in the correct combinations and moving at a wall which is required to test the algorithm	The character will accelerate in the direction held until the maximum speed is reached or the direction is released	Passed	
Boundary	Pressing opposing directions – A and D, S and W	This is possible input that a player might momentarily input as they switch directions, but since it is created from both valid inputs, but results in a different output, it should also be tested.	Combining these keys should result in no movement since they lead to movement in opposing directions	Passed	
Erroneous	Pressing	Keys not used	The player	Passed	

	another key (e.g. M)	for movement should not produce any movement	will remain still		
--	-------------------------	---	----------------------	--	--

With the original movement problem solved, I could now move onto looking as this is equally integral to the player's movement set, and therefore this might break other sections of the code. Different parts of looking need to interact with different objects. Looking side-to-side can be done by rotating the player's body on the y-axis (vertical axis) as one would do similarly in reality, however, looking up and down is done by moving your head which is separate from the body. Similarly, the whole body should not rotate when looking up and down, and only the camera should move up and down.

```
using UnityEngine;
using UnityEngine.InputSystem;
using UnityEngine.SceneManagement;

public class PlayerLook : MonoBehaviour
{
    public PlayerManager manager;
    private InputAction lookAction;
    [SerializeField] private Rigidbody playerBody;
    [SerializeField] private float mouseSensitivity;
    private float verticalRotation = 0f;

    void Start()
    {
        lookAction = manager.GetComponent<PlayerManager>().inputs.Player.Look;
    }

    // Update is called once per frame
    void Update()
    {
        Look();
    }

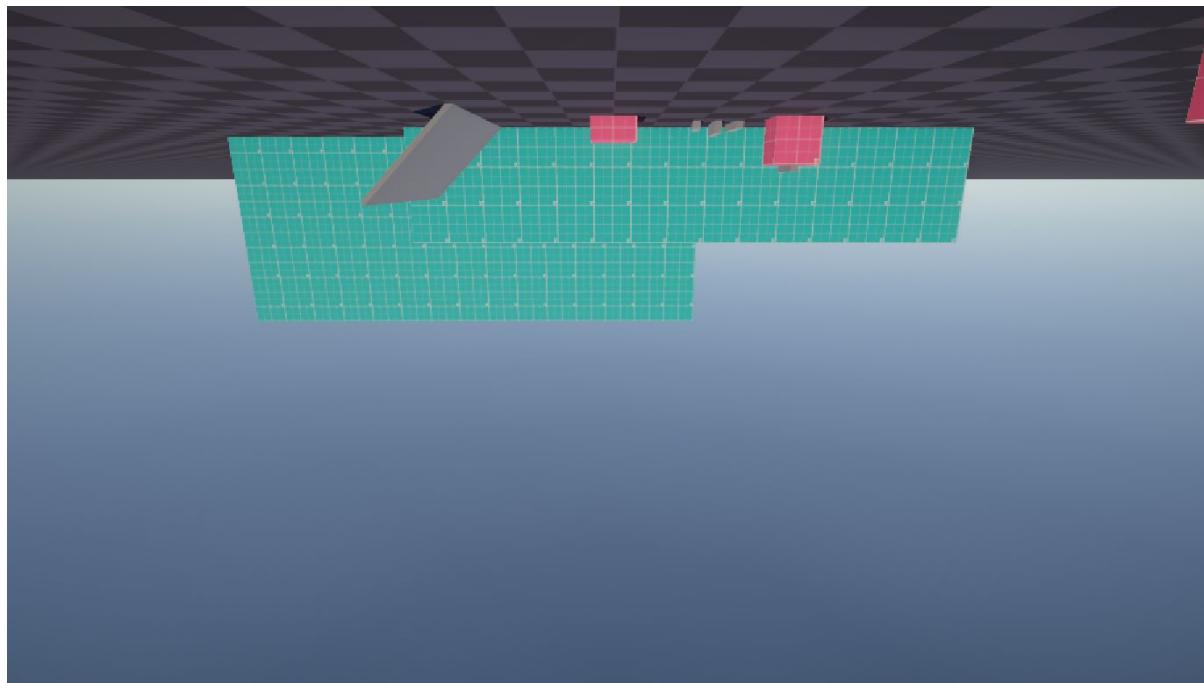
    void Look(){
        Vector2 lookMotion = lookAction.ReadValue<Vector2>() * mouseSensitivity / 20;
        playerBody.rotation = playerBody.rotation * Quaternion.Euler(Vector3.up * lookMotion.x);

        verticalRotation -= lookMotion.y;
        transform.localRotation = Quaternion.Euler(verticalRotation, 0f, 0f);
    }
}
```

The only purpose of Start() is to retrieve the look action from the manager. The rest of the looking code exists within Update, since it needs to be updated every frame. The look action is a 2D Vector, containing the change horizontally and vertically. The horizontal direction is mapped directly to the vertical rotation of the body, while the

vertical direction does not already have a value storing its rotation, so we need to add this in manually. Since this script is placed onto the camera object, transform.localRotation is used instead of referencing the camera directly to save on memory.

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Moving the mouse up and down	This is the entire input base that the player should be using for looking around	The character will look around normally	Failed: the mouse went off the screen	The cursor needs to be locked to the game window
Boundary	Moving the mouse really far	This is to check that there is no snapping in the movement and that the system is capable of large movement values	The character will look around really fast	Failed: the player was able to look upside down, which can disorient them	This might be caused by not stopping the angle from going past 90 degrees up and down
Erroneous	No movement	It is important to check that the system does not continue drifting while the player is not moving the mouse	The camera will be still	Passed	



(The player looking upside down)

There was an unforeseen error where the player was able to look past the top and bottom and continue looking upside down. This is most likely caused by the fact that the vertical rotation was not clamped between -90 and 90 degrees. This can be fixed via this line below:

```
verticalRotation -= lookMotion.y;
verticalRotation = Mathf.Clamp(verticalRotation, -90f, 90f);
transform.localRotation = Quaternion.Euler(verticalRotation, 0f, 0f);
```

Now, this is clamped between -90° and 90° to make sure that the player does not continue looking and end up backwards, theoretically breaking their neck.

Valid	Moving the mouse up and down	This is the entire input base that the player should be using for looking around	The character will look around normally	Passed	
-------	------------------------------	--	---	--------	--

The next error now needs to be solved: the player's mouse continues to move off the screen since it is not constrained within the window. This is a distraction and inconvenient when the player clicks, so Cursor.lockState is set to Locked.

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.InputSystem;

public class PlayerLook : MonoBehaviour
{
    public GameObject manager;
    private InputAction lookAction;
    [SerializeField] private Rigidbody playerBody;
    [SerializeField] private float mouseSensitivity;
    private float verticalRotation = 0f;

    void Start()
    {
        Cursor.lockState = CursorLockMode.Locked;
        lookAction = manager.GetComponent<PlayerManager>().inputs.Player.Look;
    }

    // Update is called once per frame
    void Update()
    {
        Vector2 lookMotion = lookAction.ReadValue<Vector2>() * mouseSensitivity / 20;
        playerBody.rotation = playerBody.rotation * Quaternion.Euler(Vector3.up * lookMotion.x);

        verticalRotation -= lookMotion.y;
        verticalRotation = Mathf.Clamp(verticalRotation, -90f, 90f);
        transform.localRotation = Quaternion.Euler(verticalRotation, 0f, 0f);
    }
}

```

Boundary	Moving the mouse really far	This is to check that there is no snapping in the movement and that the system is capable of large movement values	The character will look around really fast	Passed	
----------	-----------------------------	--	--	--------	--

Now that the system is finished, I tested it again with all the tests again, to make sure none of the fixes broke anything.

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Moving the mouse up and down and side to side	This is the entire input base that the player should be using for looking around	The character will look around normally	Passed	
Boundary	Moving the mouse really far in any direction	This is to check that there is no snapping in the movement and that the system is capable of large movement values	The character will look around really fast	Passed	
Erroneous	No movement	It is important to check that the system does not continue drifting while the player is not moving the mouse	The camera will be still	Passed	

Looking alone worked fine, and now I needed to test it with movement together.

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Moving the mouse up and down while moving in a circle	This combines all the looking actions with all the movement options and is also covers the most typical of a player's movement	The character will look around and the player will move around relative the current looking direction.	Failed: the player moved around independent of their facing angle.	Most likely caused by the movement script writing global direction instead of local directions.

		style			
Boundary	Moving the mouse really far while moving in a circle	This is to check that there is no snapping in the movement and that the system is capable of large movement values, while maintaining movement	The character will look around really fast and will move accordingly	Failed: Encountered the same error as the valid data	
Erroneous	No movement	It is important to check that the system does not continue drifting while the player is not moving the mouse	The camera will be still, and the player will not move	Passed	

This error seems to be caused because the code was adding force to the rigidbody according to the global axis, and not the local axis which is what I thought it was doing. If it was adding to the local axis, then everything would work fine, since the rotation to the body would be applied after. To remedy this, we need to rotate the target (which is the player's input) to face the body's current direction. The player's body rotation is stored as a quaternion, which can be used to rotate a vector.

```
Vector3 target = player.rotation * new Vector3(inputDirection.x, 0, inputDirection.y);
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Moving the mouse up and down while moving in a circle	This combines all the looking actions with all the movement options and is also covers the most typical of a	The character will look around and the player will move around relative the current looking direction.	Passed	

		player's movement style			
Boundary	Moving the mouse really far while moving in a circle	This is to check that there is no snapping in the movement and that the system is capable of large movement values, while maintaining movement	The character will look around really fast and will move accordingly	Passed	
Erroneous	No movement	It is important to check that the system does not continue drifting while the player is not moving the mouse	The camera will be still, and the player will not move	Passed	

Since all the tests passed, I asked my stakeholders about the system and see if anything else was needed to be added.

**Stakeholder:** Hmm, playing this version of the movement is alright, but I feel like the player doesn't turn around fast enough, nor do they start moving fast enough. Is it possible to make them move faster at the beginning or when turning around?

To allow the player to do these things, we need to calculate the how different the current movement and the opposing movement is, which can be done with the dot product. If both vectors have a magnitude of 1 (which is true for all normalized vectors), then the dot product returns a value that is negative when the vectors are in directions that oppose each other. This can be used with an if statement to boost the acceleration of the player. To solve the player moving slowly at the beginning, we can work out check if the speed is lower than a certain amount and hence double the acceleration.

```

private void Movement() {
    float maxSpeed = MaxSpeed();
    float acceleration = CalculateAccelerationMultiplier();
    Vector3 target = new Vector3(inputDirection.x, 0, inputDirection.y);
    float alignment = Vector3.Dot(surfaceVelocity.normalized, target.normalized);

    if (surfaceVelocity.magnitude > maxSpeed) {
        acceleration *= surfaceVelocity.magnitude / maxSpeed;
    }
    Vector3 direction = target * maxSpeed - surfaceVelocity;
    float directionMag = direction.magnitude;
    direction = Vector3.ProjectOnPlane(direction, groundNormal).normalized * directionMag;

    if (direction.magnitude < 0.5f) // If moving very slowly
    {
        acceleration *= direction.magnitude / 0.5f;
    }

    if (alignment <= 0){ // If attempting to move in a wildly opposite direction
        acceleration *= 2;
    }

    Vector3 targetMovement = direction.normalized * acceleration;
    targetMovement -= direction * FrictionMultiplier();

    player.AddForce(targetMovement * Time.deltaTime, ForceMode.VelocityChange);
}

```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Moving the player from rest	This is one of the requirements of what the stakeholder recently requested and hence needs to be checked appropriately	The player will accelerate faster than before	Passed	
Valid	Moving the player in a completely opposing direction than they are currently moving	This is another requirement that the stakeholder recently requested	The player should change to the opposing direction much faster than before	Passed	
Erroneous	No change in the current movement	It is important to check that the system has not	The player will continue moving in the same way	Passed	

		changed how it performs			
--	--	-------------------------	--	--	--

Next, I decided it would be better to make the game more performant by utilising the built in Physics system to tell the player controller when it is grounded or not. When a collider collides with something, Unity automatically calls `OnCollisionEnter()` and `OnCollisionStay()` on all scripts that the object has attached. When this collider stops colliding with an object, Unity automatically calls `OnCollisionExit()` in the same way. All these functions have a list of the collision points relative to the centre of the player passed in as a parameter. We can simply go through each of these and figure out if the contact's normal is far shallow enough to be a floor.

```
using UnityEngine;

public class BodyDetection : MonoBehaviour
{
    public PlayerMovement bodyScript;

    private void OnCollisionStay(Collision collision) {
        bodyScript.CollisionDetected(collision);
    }
    private void OnCollisionExit(Collision collision) {
        bodyScript.CollisionDetected(collision);
    }
}
```

(This script is attached to the body directly ^)

```
public void CollisionDetected(Collision collision){ // This function is called externally by the body
    if (collision.contacts.Length > 0) {
        foreach (ContactPoint contact in collision.contacts) {
            float slopeAngle = Vector3.Angle(contact.normal, Vector3.up);

            if (slopeAngle <= maxSlope){
                isGrounded = true;
                groundNormal = contact.normal;
                break;
            }
        }
    } else {
        isGrounded = false;
        groundNormal = Vector3.up;
    }
}
```

If there are no contacts, then player is in midair, and hence not grounded at all. To test this, the player will jump.

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the	When the	isGrounded	Passed	

	jump key	player jumps, they are no longer in the air, and I can check the state of isGrounded	will be false.		
Boundary	Standing on the ground next to a wall	One of the contacts will be on the ground, while another is on a wall, and the game may get confused with this	isGrounded will be true	Passed	
Erroneous	Pressing another key (e.g. M)	This should keep the player on the ground	isGrounded will be true	Passed	

With this, I have made sure that game is running much more efficiently, and now I can add a further feature of crouching. Crouching is an action where the player will both move at a slower speed, and while also shrink in height. To shrink the player's height, we can manually change the player collider's height to a smaller size that is more appropriate.

```
private void SetPlayerDimensions(float height, float radius){
    playerCollider.height = height;
}

private void Crouch(){
    if (movementState == PlayerMovementState.crouching){
        SetPlayerDimensions(crouchingHeight);
    } else {
        SetPlayerDimensions(standingHeight);
    }
}
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the crouch key	This is input that the player will use, and hence needs to be tested appropriately	The player's height will be shrunk	Partial: the player did decrease in height, but in mid air	The player shrunk around their centre instead of

					their base
Erroneous	Pressing another key (e.g. M)	An input not related to the test should not affect the test	The player will remain at their standing height	Passed	

Upon crouching, the player does indeed shrink in size, but unfortunately, they shrink in midair and end up falling to the ground. This can be solved by finding the distance required to move the player down after the shrink, or back up after they stop crouching, and then taking this away from the player's current y-position. This value needs to be halved since I only need the part of the shrunk height that is downwards from the centre of the player. To do all of this, I also need to store the height before the change

```
private void SetPlayerDimensions(float height){
    float previousHeight = playerCollider.height;
    playerCollider.height = height;
    player.transform.position -= new Vector3(0, (previousHeight - height)/2, 0);
}
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the crouch key	This is input that the player will use, and hence needs to be tested appropriately	The player's height will be shrunk	Passed	

Next, I can implement the ability to crouch with either a toggle crouch or with a crouch key that must be held. Following the algorithm that was designed earlier and testing as such:

```

private bool CrouchControlState(){
    holdCrouch = playerInputs.Player.HoldCrouch.inProgress;
    if (playerInputs.Player.ToggleCrouch.triggered){
        if (toggleCrouch){
            toggleCrouch = false;
        } else {
            toggleCrouch = true;
        }
    }
    if (holdCrouch){
        lastHoldCrouchState = true;
        return true;
    } else {
        if (lastHoldCrouchState){
            toggleCrouch = false;
        }
        lastHoldCrouchState = false;
        return toggleCrouch;
    }
}

```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the toggle crouch key	Each input must be tested separately to verify that they work	The player will crouch	Passed	
Valid	Holding the hold crouch key	Each input must be tested separately to verify that they work	The player will crouch for the duration that the key is held	Passed	
Boundary	Pressing the toggle crouch key and then the hold crouch key, and then releasing it	As was specified in the first interview, the held crouch must take priority over	The player will crouch and then uncrouch when the hold crouch key is released	Passed	
Erroneous	Pressing another key (e.g. M)	An input not related to the test should not affect the test, and	The player will remain at the same height	Passed	

		hence checks that the input system is still working as expected			
--	--	---	--	--	--

However, looking at the code, I realised that the toggleCrouch if statement was simple enough to instead use the C# ternary operator instead, which can compress 1 line if else statements.

```
toggleCrouch = toggleCrouch ? false : true;
```

Taking a look at the code from this view, however, I realised that it was simpler to invert the current state of toggleCrouch, increasing the efficiency of the algorithm even further.

```
toggleCrouch = !toggleCrouch;
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the toggle crouch key	Each input must be tested separately to verify that they work	The player will crouch	Passed	
Valid	Holding the hold crouch key	Each input must be tested separately to verify that they work	The player will crouch for the duration that the key is held	Passed	
Boundary	Pressing the toggle crouch key and then the hold crouch key, and then releasing it	As was specified in the first interview, the held crouch must take priority over	The player will crouch and then uncrouch when the hold crouch key is released	Passed	
Erroneous	Pressing another key (e.g. M)	An input not related to the test should not affect the test, and hence checks that the input system is still	The player will remain at the same height	Passed	

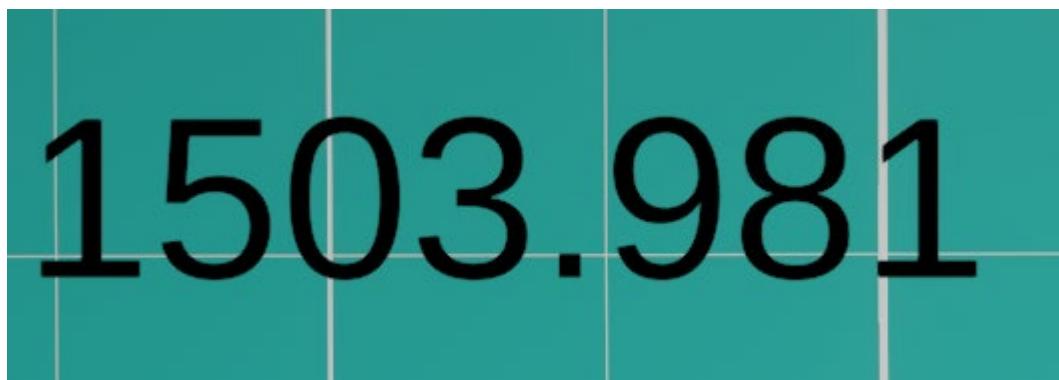
		working as expected			
--	--	---------------------	--	--	--

## Miscellaneous

To measure the performance of the game, I decided to add an on-screen element to measure how many frames are being run per second (FPS). The higher the FPS of the game, the better it is running since less calculations are done per frame, and hence the easier it is to run. The simplest way to do this was to follow this equation  $f = \frac{1}{T}$ , where the frequency is the FPS, and T (the time period) is how long each frame took to run. This value can be updated every frame.

```
public TMP_Text text;

void Update(){ // Displays the current framerate
    float displayVal = 1f/Time.deltaTime;
    text.text = displayVal.ToString();
}
```



(The FPS counter in the game)

However, this noticeably has a problem – the value changes too quickly to get an accurate reading of what it is and as such it might be better to use a method which averages the FPS across the game.

```
int number = 0;
float currentAvg = 0;
float UpdateAverage(float currentFPS){ // Optional version that averages the fps across the whole game
    number++;
    currentAvg += (currentFPS - currentAvg)/number;
    return currentAvg;
}
```

This version instead stores how many frames have gone by, and what the current average is, it can then work out how much the average would change based on the current FPS and then update the average.

While this works well, as the game goes on longer and longer, the average becomes less sensitive to the current FPS as each progressive value has less and less of an impact on the whole. As such, I decided to scrap this version since the player is expected to play the game for periods of time longer than 10 minutes, whereas the average FPS becomes useless at around 30 seconds. Instead, to help with debugging the game, I added an option for the current FPS to be logged to the debug file instead.

```
public TMP_Text text;
public bool log;

void Update(){ // Displays the current framerate
    float displayVal = 1F/Time.deltaTime;
    text.text = displayVal.ToString();
    if (log){
        Debug.Log("FPS: " + displayVal.ToString());
    }
}
```

While this works ok, I noticed a noticeable decrease in the FPS when this option was used, and the player's original framerate was extremely high. This makes sense as writing to a file is quite the intensive operation, and as such this logging facility should be used as sparingly as possible, but exists as a useful option when the game's FPS is lower, since at FPS lower than 200, the IO operation required to write to the debug file takes less time than the rest of the game by a long margin.

## End of prototype tests

### *Strafing*

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the W/A/S/D keys alone and in combinations while moving at a wall	These keys allow the player to move in 8 directions once used in the correct combinations and moving at a wall which is required to test the algorithm	The character will accelerate in the direction held until the maximum speed is reached or the direction is released	Passed	
Boundary	Pressing opposing	This is possible input	Combining these keys	Passed	

	directions – A and D, S and W	that a player might momentarily input as they switch directions, but since it is created from both valid inputs, but results in a different output, it should also be tested.	should result in no movement since they lead to movement in opposing directions		
Erroneous	Pressing another key (e.g. M)	Keys not used for movement should not produce any movement	The player will remain still	Passed	

*Looking*

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Moving the mouse up and down	This is the entire input base that the player should be using for looking around	The character will look around normally	Passed	
Boundary	Moving the mouse really far	This is to check that there is no snapping in the movement and that the system is capable of large movement values	The character will look around really fast	Passed	
Erroneous	No movement	It is important to check that the system does not	The camera will be still	Passed	

		continue drifting while the player is not moving the mouse			
--	--	--	--	--	--

*Boosting*

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing Q	This should be the only key used for boosting, and hence should respond appropriately	The player's speed increases by a huge amount	Passed	
Erroneous	Pressing another key (e.g. M)	Keys not used for boosting should not activate it	The player will remain at their current speed	Passed	

*Jumping*

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the jump key	As the only input being used to jump, it is important that it works	The character will move upwards, then move back down	Passed	
Erroneous	Pressing another key (e.g. M)	Since there is only one input, nothing else should cause the player to jump, and if there is, that means something is significantly broken	The player will remain still	Passed	

*Crouching*

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the crouch key	This is input that the player will use, and hence needs to be tested appropriately	The player's height will be shrunk	Passed	
Erroneous	Pressing another key (e.g. M)	An input not related to the test should not affect the test	The player will remain at their standing height	Passed	

*Different crouch keys*

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the toggle crouch key	Each input must be tested separately to verify that they work	The player will crouch	Passed	
Valid	Holding the hold crouch key	Each input must be tested separately to verify that they work	The player will crouch for the duration that the key is held	Passed	
Boundary	Pressing the toggle crouch key and then the hold crouch key, and then releasing it	As was specified in the first interview, the held crouch must take priority over	The player will crouch and then uncrouch when the hold crouch key is released	Passed	
Erroneous	Pressing another key (e.g. M)	An input not related to the test should not affect the test, and hence checks that the input system is still	The player will remain at the same height	Passed	

		working as expected			
--	--	---------------------	--	--	--

*Looking and strafing combined tests*

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Moving the mouse up and down while moving in a circle	This combines all the looking actions with all the movement options and is also covers the most typical of a player's movement style	The character will look around and the player will move around relative the current looking direction.	Passed	
Boundary	Moving the mouse really far while moving in a circle	This is to check that there is no snapping in the movement and that the system is capable of large movement values, while maintaining movement	The character will look around really fast and will move accordingly	Passed	
Erroneous	No movement	It is important to check that the system does not continue drifting while the player is not moving the mouse	The camera will be still, and the player will not move	Passed	

## Interview

Me: I've finished implementing the basic movement options and I've confirmed that there are no errors with them. Should I now move onto the entity system and the player's ability to damage them?

SillyAustralian: First, let me try the game so far. If everything is ok, then you can move onto those features.

[Henry Masters proceeded to play the prototype in its current state]

SillyAustralian: Ok, everything is fine here. It is ok to move onto the weapon interaction and entity system. As previously discussed, due to the setbacks that have occurred, you do not need to implement anything other than a gun for the weapon system, and you do not need to make the entities communicate with any other entity, including the manager.

Stakeholder Signature:



## Prototype 2

### Interacting with weapons

Weapon interaction will begin with 1 weapon.

To begin interacting with weapons, first I need to make a weapon. Weapon will be an abstract class that contains information that is applicable or can be made applicable to all types of weapons and Fire() which will be used to activate the weapon. The properties that are applicable are:

- The type of the weapon, which will function as an ID
- The total ammunition that the weapon has
- The amount left in the clip if applicable
- The capacity of the weapon's clip

Weapon is a class that must be made abstract, and hence all the internal properties will be abstract as well. These will have default values, since not every child class will need to overwrite the values.

```
abstract public int TotalAmmo => 1;
abstract public int AmmoInClip => 'Weapon.TotalAmmo.get' cannot declare a body because it is marked abstract (CS0500)
abstract public void Fire();
```

However, attempting to do this will cause an error since abstract functions (C# properties act as variables that are functions under the surface) cannot have anything within them. As such, these need to be defined as virtual instead, which is a special keyword in C# that allows a function to be overridden while containing a base implementation.

```
using UnityEngine;

public abstract class Weapon : MonoBehaviour
{
    virtual public string WeaponType {get;}
    virtual public int TotalAmmo {get; set;}
    virtual public int AmmoInClip {get; set;}
    virtual public int ClipCapacity {get; set;}
    abstract public void Fire();
}
```

Next, a gun will have a type of bullet that it will fire, that will deal a certain amount of damage, and it will also have a barrel where the bullet is fired from. Importantly, the bullet itself is the thing that will be dealing the damage, not the gun itself. This way, the class can be reused for other weapons which will keep the codebase smaller and easier to maintain, since there will be only 1 bullet that can deal damage to entities.

Bullet is a class that will have 1 purpose: to check whether it will hit an entity, and if it does, to deal damage to it. As such, its contents are relatively simple in comparison:

```
public class Bullet : MonoBehaviour
{
    [SerializeField] private float damage = 1f;
    private void Update()
    {
        RaycastHit hitInfo;
        if (Physics.Raycast(origin: transform.position, direction: transform.rotation * Vector3.up, hitInfo: out hitInfo)){
            GameObject hit = hitInfo.collider.gameObject;
            Entity entityClass = hit.GetComponent<Entity>();
            if (entityClass != null){
                hit.SendMessage("Damage", damage);
            }
        }
        Destroy(gameObject);
    }
}
```

Once the bullet has performed its task, it destroys itself, keeping memory consumption low, and making the game as easy to run as possible. To fire said bullet, the gun will first check there is enough ammunition remaining, and if so, it will instantiate a bullet entity. Afterwards, it will then subtract 1 from both TotalAmmo and AmmoInClip.

```
override public void Fire(){ // Instantiates a bullet at the gun's barrel traveling outwards
    Instantiate(bullet, barrel.transform.position, barrel.rotation);
    Debug.Log("Fired " + gameObject.name);
    TotalAmmo -= 1;
    AmmoInClip -= 1;
}
```

On the other side of the system, the player's input needs to be read and passed through to the weapon. This is done by subscribing a pass-through method to the correct input action, which upon activation, will call the relevant action on the gun.

```
private void Start()
{
    manager = PlayerManager.Instance;
    playerInputs = manager.inputs;
    playerInputs.Player.Fire.performed += Fire;
    playerInputs.Player.Reload.performed += Reload;
}
private void Fire(InputAction.CallbackContext inputType)
{
    weapon.GetComponent<Gun>().Fire();
}
private void Reload(InputAction.CallbackContext inputType)
{
    weapon.GetComponent<Gun>().Reload();
}
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the fire button while there is enough ammunition remaining	This is the most common action the weapon will encounter and therefore is suitable as a valid test	The gun will fire 1 bullet	Passed	
Boundary	Pressing the fire button while there is no ammunition remaining	It is important the system still acts accordingly when there is no ammunition in the clip remaining	The gun will not fire	Passed	
Erroneous	Pressing another key	Any inputs not related to the	The gun will not fire	Passed	

	(e.g. M)	system should not affect it			
--	----------	-----------------------------------	--	--	--

However, once the gun runs out of ammunition, the player needs to reload it. Since the player has limited ammunition, this needs to be taken into account when reloading. The player could also attempt to reload in the middle of a clip, and hence only the excess should be topped up.

```
public virtual void Reload(){ // Reloads the clip
    int excess = ClipCapacity - AmmoInClip;
    if (TotalAmmo >= excess){ // Check if there is enough ammo to fill the clip back to full
        TotalAmmo -= excess;
        AmmoInClip = ClipCapacity;
    } else{ // If there isn't enough ammo remaining to make the clip full
        AmmoInClip = TotalAmmo;
        TotalAmmo = 0;
    }
}
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the reload button when there is no ammunition left in the clip, and there is enough ammunition remaining in total	This is the next most common action the weapon will encounter and therefore is suitable as a valid test	The gun will reload back to full	Passed	
Boundary	Pressing the reload button when there is ammunition left in the clip	It is important the system still acts accordingly when the player reloads before they need to	The gun will reload back to full and only take as much ammo as needed	Passed	
Erroneous	Pressing another key (e.g. M)	Any inputs not related to the system should not affect it	The gun will not reload	Passed	

To implement zooming, the game needs to take control of the camera's field of view. If the camera has a lower field of view, it can see less things at one time, but since this lower number of objects takes up more space on the screen, it gives the effect of having zoomed in. Meanwhile, doing the opposite will have the effect of zooming out. By binding zoom to the action being triggered, the player can toggle between zooming in and out, by setting this value on the camera.

```
private void Zoom(InputAction.CallbackContext inputType){
    if (isZooming){
        playerHead.fieldOfView = zoomFov; // Set the camera's fov to the zoom fov
    } else {
        playerHead.fieldOfView = fov; // Reset the camera's fov
    }
}
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the zoom button and then pressing it again	While the player may not use this action a lot, this is the only way to enable it and disable it, therefore making this suitable as a valid test	The player's view will zoom in – focus on a smaller area, and then back out	Passed	
Valid	Pressing the zoom button and then moving the mouse around	The player will most likely want to look around while zooming in as well, hence this must be tested as well	The player's view will zoom in and then move around appropriately	Failed – the view was uncontrollable from moving too fast	The sensitivity needs to change when
Erroneous	Pressing another key (e.g. M)	Any inputs not related to the system should not affect it	The player's view will remain the same	Passed	

When the player had zoomed in, it became much harder to control the view accurately, and as such, it is necessary to turn down the sensitivity of the mouse while

```

private void Zoom(InputAction.CallbackContext inputType){
    if (isZooming){
        playerHead.fieldOfView = zoomFov; // Set the camera's fov to the zoom fov
        lookScript.sensitivity = lookScript.zoomSensitivity; // Lower the sensitivity
    } else {
        playerHead.fieldOfView = fov; // Reset the camera's fov
        lookScript.sensitivity = lookScript.normalSensitivity; // Reset the sensitivity
    }
}

```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the zoom button and then moving the mouse around	The player will most likely want to look around while zooming in as well, hence this must be tested as well	The player's view will zoom in and then move around appropriately	Passed	

Extending this current system to use 4 weapons instead of 1, as was requested can be done by creating 4 weapon slots like below and a tracker of which slot is currently active:

```

public GameObject weaponSlot0 = null;
public GameObject weaponSlot1 = null;
public GameObject weaponSlot2 = null;
public GameObject weaponSlot3 = null;
public int activeWeaponSlot = 0;

```

This can then be paired with a switch statement to access each slot.

```

private GameObject SelectFireWeapon(){
    switch(activeWeaponSlot){
        case 0:
            return weaponSlot0;
        case 1:
            return weaponSlot1;
        case 2:
            return weaponSlot2;
        case 3:
            return weaponSlot3;
        default:
            return weaponSlot0;
    }
}

```

Swapping between slots requires changing activeWeaponSlot and the standard key that is used to do this is the scroll wheel. The scroll wheel returns a float value, although this is only -1.0 and 1.0, indicating the direction the scroll wheel is moving in. Therefore, this can be casted to an integer with no backlash. The current weapon needs to be deactivated, and then the newly selected weapon needs to be enabled. If the player continues scrolling in the same direction, the value should wrap around to the start.

```
private void SwapWeapon(InputAction.CallbackContext inputType)
{
    int direction = (int)inputType.ReadValue<float>();
    weaponSlots[activeWeaponSlot].SetActive(false);
    weaponSlots[activeWeaponSlot].transform.position = weaponStow.position;
    activeWeaponSlot = Wrap(activeWeaponSlot + direction, weaponSlots);
    weaponSlots[activeWeaponSlot].transform.position = weaponHold.position;
    weaponSlots[activeWeaponSlot].SetActive(true);
}
private int Wrap(int value)
{
    return value % 4;
}
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Scrolling the scroll wheel in either direction	The player will want to change the current weapon based on the scenario and therefore this is suitable as a valid test	The selected weapon will change relative to the direction the player scrolled in	Failed: When scrolling backwards, the weapon did not change once it reached the first weapon	MOD likely does not work the same when comparing a negative number and a positive one
Boundary	Scrolling very fast in either direction	It is important the system still acts accordingly when the player may put it under stress	The currently selected weapon should change accordingly	Failed: Same as the above error	
Boundary	Scrolling when all the weapon slots are not filled	It can occur that all the player's weapons slots are not	The system should continue wrapping with the	Failed: Reached a NullReference Exception when the	Nothing is stopping Wrap() from accessing

		filled and as such the scrolling should still respond correctly, even though this is a rare case.	remaining weapons and change the weapons accordingly.	empty slot was reached and the same error as the above when scrolling in reverse	the empty variables
Erroneous	Pressing another key (e.g. M)	Any inputs not related to the system should not affect it	The current weapon will remain selected	Passed	

The problem caused by scrolling in reverse is caused due to the MOD operator, as  $-1 \text{ MOD } 4$  does not loop round to 3. To counteract this, you must add on the number that you are performing MOD with, which in this case, is 4. This offsets the negative number and is cancelled out for the positive numbers by MOD.

```
private int Wrap(int value)
{
    return ((value % 4) + 4) % 4;
}
```

However, this only works when all the player's weapon slots are filled. If any weapon slot is empty, such as in a tutorial or elsewhere, this throws a NullReferenceException. The next way of creating 4 weapon slots could be done with an array since the maximum number of weapons has already been predetermined.

```
private void SwapWeapon(InputAction.CallbackContext inputType)
{
    int direction = (int)inputType.ReadValue<float>();
    weaponSlots[activeWeaponSlot].SetActive(false);
    weaponSlots[activeWeaponSlot].transform.position = weaponStow.position;
    activeWeaponSlot = Wrap(activeWeaponSlot + direction, weaponSlots);
    if (SelectFireWeapon() == null)
    {
        activeWeaponSlot = Wrap(activeWeaponSlot - direction, weaponSlots);
    }
    weaponSlots[activeWeaponSlot].transform.position = weaponHold.position;
    weaponSlots[activeWeaponSlot].SetActive(true);
}
```

Adding an extra check for if the returned output of SelectFireWeapon() is null (meaning that there is no weapon in that slot) does indeed work for when there are only 2 or 3 weapons in all the slots. However, when there is only 1 or even no weapons equipped, this fails again, and adding an extra exception for when there is only 1 weapon is not the correct approach either. Instead, the best solution is to have weaponSlots be a list, and to stop adding elements to it once it reaches the limit which is 4. This way, Wrap() can be modified to wrap with the current length.

```
private int Wrap(int value, List<GameObject> list) {
    return ((value % list.Count) + list.Count) % list.Count; // Wrap both around negatively and positively
}
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Scrolling the scroll wheel in either direction	The player will want to change the current weapon based on the scenario and therefore this is suitable as a valid test	The selected weapon will change relative to the direction the player scrolled in	Passed	
Boundary	Scrolling very fast in either direction	It is important the system still acts accordingly when the player may put it under stress	The currently selected weapon should change accordingly	Passed	
Boundary	Scrolling when all the weapon slots are not filled	It can occur that all the player's weapons slots are not filled and as such the scrolling should still respond correctly, even though this is a rare case.	The system should continue wrapping with the remaining weapons and change the weapons accordingly.	Passed	
Erroneous	Pressing another key	Any inputs not related to the	The current weapon will	Passed	

	(e.g. M)	system should not affect it	remain selected		
--	----------	-----------------------------------	--------------------	--	--

To go hand in hand with this, adding weapons to the current arsenal works very similarly. First a RayCast is performed, and if the hit object has the “Weapon” tag, then that weapon is added to the arsenal. If all the weapon slots are full, then the weapon in the current slot is replaced. On the other hand, if there is space remaining, then activeWeaponSlot is pointed to the next available slot, and the weapon is simply added to the list.

```

private int GetSwapSlot(out bool wasUsed)
{
    if (weaponSlots.Count < 4){
        wasUsed = false;
        return activeWeaponSlot + 1;
    }
    wasUsed = true;
    return activeWeaponSlot; // Return the current slot if there isn't an empty one
}
private void Interact(InputAction.CallbackContext inputType)
{
    RaycastHit hit;
    if (Physics.Raycast(origin: playerHead.transform.position, direction: playerHead.transform.rotation * Vector3.forward, hitInfo: out hit)){
        Debug.Log("Raycast hit " + hit.collider.gameObject.name);
        if (hit.collider.gameObject.tag == "Weapon") // Verify that the hit is a weapon
        {
            AddWeapon(hit.collider.gameObject);
        }
    }
}
public void AddWeapon(GameObject hit){
    Debug.Log("weapon interaction");
    bool wasUsed;
    int temp = GetSwapSlot(out wasUsed);
    if (wasUsed){ // remove the current weapon
        weaponSlots[activeWeaponSlot].transform.SetParent(null, true);
        weaponSlots[activeWeaponSlot].GetComponent<Rigidbody>().isKinematic = false;
        weaponSlots[activeWeaponSlot].GetComponent<Rigidbody>().detectCollisions = true;
    } else if (!weaponSlots.Contains(hit)){ // Add the new weapon
        weaponSlots.Add(hit);
    }
    activeWeaponSlot = temp;
    // Format the new weapon to the appropriate Locations and properties
    weaponSlots[activeWeaponSlot].transform.SetParent(playerHead.transform, true);
    weaponSlots[activeWeaponSlot].GetComponent<Rigidbody>().isKinematic = true;
    weaponSlots[activeWeaponSlot].GetComponent<Rigidbody>().detectCollisions = false;
    weaponSlots[activeWeaponSlot].transform.position = weaponHold.position;
    weaponSlots[activeWeaponSlot].transform.rotation = weaponHold.rotation;
}

```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Adding a weapon while there is space available	The player will want to change the current weapon based on the scenario and therefore this is suitable as a valid test	The weapon will be added to the list with no other weapon removed, and will swap to that weapon	Passed	
Valid	Adding a	The player will	The weapon	Passed	

	weapon while there is no space available	want to change the current weapon based on the scenario, and this is the more common way to acquire a weapon	will replace the current weapon		
Erroneous	Pressing another key (e.g. M)	Any inputs not related to the system should not affect it	The current weapon will remain selected	Passed	

## Entities (Damage)

The most basic feature of this system is that it handles the health of the object that it is attached to. All entities will have a current health value that upon being initialised will be set to the maximum health. All entities will have 3 base functions:

- **Damage()** – Mentioned above, Damage() is what is called by all objects that need to damage an entity. Entities will also have a defence multiplier that will decrease the amount of damage they take.
- **Healing()** – These allow the entity to heal. While it won't be used on most entities, it is still useful to have underneath in case SillyAustralian wishes to use it for any special entity types
- **Kill()** – This is absolutely required to remove the entity once their health has reached zero. It is also useful for debugging purposes and for special events if required. Leaving this as a separate function leaves as much functionality remaining for any potential uses or upgrades to the system.

```
using UnityEngine;

public abstract class Entity : MonoBehaviour
{
    public float MaxHealth { get; set; }
    public float Health { get; set; }
    public float DefenseMultiplier { get; set; }
    public float HealingMultiplier { get; set; }

    private void Start()
    {
        Health = MaxHealth;
    }

    public void Damage(float attack)
    {
        Health -= attack * (1 - DefenseMultiplier);
        Health = Mathf.Clamp(Health, 0, MaxHealth);
        if (Health < 0)
        {
            Kill();
        }
    }

    public void Healing(float heal)
    {
        Health += heal * (1 + HealingMultiplier);
        Health = Mathf.Clamp(Health, 0, MaxHealth);
    }

    public void Kill()
    {
        Destroy(gameObject);
    }
}
```

To appropriately test the damage, it was easiest to make a spike – an object that deals damage on contact to the entity. To do so, I can leverage the code that the bullet uses to check whether it has collided with something but modified to use `OnCollisionEnter()` instead.

```

using UnityEngine;

public class Spike : MonoBehaviour
{
    public float damage = 10;
    void OnCollisionEnter(Collision collision)
    {
        foreach (ContactPoint contact in collision.contacts){
            GameObject hit = contact.otherCollider.gameObject;
            Entity entityClass = hit.GetComponent<Entity>();
            if (entityClass != null){
                hit.SendMessage("Damage", damage);
            }
        }
    }
}

```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Running into the spike	This is a basic test for damage as well as well as verifying environmental damage	The player will take damage as is detailed	Passed	
Boundary	Running into a spike with 1 HP (or enough for the next hit to kill the player)	This will test whether the kill function works	The player will disappear from the scene, and nothing will be visible on the screen	Failed: received null reference exception	



[22:32:31] NullReferenceException: Object reference not set to an instance of an object  
 Entity.Kill (UnityEngine.GameObject obj) (at Assets/Scripts/Entities/Entity.cs:36)

This exception seems to occur because the kill function is looking for the GameObject that it is attached to. While from first glances the code seems like it should work, it turns out it is specifically looking for the Entity class to be attached to an object, which isn't attached to any class. The current one that is attached is PlayerEntity, which is a different class than Destroy() is expecting. As such, the contents of Entity.Kill() need to be copied into PlayerEntity to test Kill() appropriately.

Type	What data fits criteria	Justification	Expected	Actual	Comments

Boundary	Running into a spike with 1 HP (or enough for the next hit to kill the player)	This will test whether the kill function works	The player will disappear from the scene, and nothing will be visible on the screen	Passed	
----------	--	--	---	--------	--

Now that the base functions have been tested and verified, the next step is to make the enemies. Importantly, the secondary function of the system is to control how the enemy functions. The enemy will have 3 states: Searching, Combat, and None. The two that matter are Searching and Combat, where the enemy has tasks that they must perform.

While the enemy is in the Searching state, they will move towards a randomly decided target. Once this target has been reached, the next target will be determined and then the enemy will move towards it. If the enemy detects the player at any moment, then the enemy will switch over to the Combat state, where it will follow behind the player. If the enemy is facing the player, then it will fire the weapon that is attached to it, dealing damage to the player if the bullet strikes them.

Implementing the enemy movement can done in a similar manner to the player and will only need a few modifications to work with the slightly different input style. The bulk of the code works the same as the player's, while the only notable change being at which point it will attempt to move away from the player.

```

protected virtual void Move(){
    Vector3 distanceToTarget = target.transform.position - gameObject.transform.position;
    Vector3 targetDirection = Vector3.forward;

    if (distanceToTarget.magnitude < NearTargetRange){
        targetDirection = Vector3.back;
    } else if (distanceToTarget.magnitude > FarTargetRange){
        targetDirection = Vector3.forward;
    } // Will auto slow down once within ranges
    targetDirection = rb.transform.rotation * targetDirection;

    float acceleration = BaseMovementAcceleration;
    if (rb.linearVelocity.magnitude > MaxMoveSpeed){
        acceleration *= rb.linearVelocity.magnitude / MaxMoveSpeed;
    }
    Vector3 direction = targetDirection * MaxMoveSpeed - rb.linearVelocity;

    float directionMag = targetDirection.magnitude;
    targetDirection = Vector3.ProjectOnPlane(direction, groundNormal).normalized * acceleration;

    targetDirection -= targetDirection * frictionMultiplier;
    rb.AddForce(targetDirection * Time.deltaTime, ForceMode.VelocityChange);
}

```

To test the system, I will be using a small sphere that does not have a collider on it to visually identify if the entity accurately reaches the target.

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	The target is outside NearRange	As the most common distance for the target from the player it should work as the valid test	The entity will approach the target and then settle at some point between NearRange and FarRange	Failed: The entity shoots past either range and returns back and oscillates like so	The entity is unable to slow itself down appropriately and dampen as it approaches its target
Boundary	The target is inside NearRange	Being very close to the target can cause the system to apply too	The entity should move away and then return to an appropriate	Failed: settled into the same issue as above	Same as above

		much force and cause the entity to fly off.	distance quickly		
Boundary	The entity is at the target	The entity reaching the target is a required part of its movement while it is in the Searching state, and as such will need to be tested	The entity will not move	Passed	
Erroneous	No target is set	This is a test for a potential error in case the target that the entity is tracking disappears and hence should be handled appropriately	The entity will not move	Passed	
Erroneous	The target is out of reach of the entity	It is incredibly likely that the player will get somewhere that the entity will be unable to reach, and as such it should be able to handle with this	The entity will attempt to reach as close to the target as possible	Passed	

Once again, researching an appropriate way to fix the problem landed me upon a better solution, of which was used in many industries, such as robotics and machinery. To work out whether this was the better option, I needed to consult my stakeholder.

*Interview*

Me: I would like to change how the enemy works as the current solution seems to be inefficient and as it is early enough, changing it to a much better solution would benefit in the long run, while solving the current problem in a much better manner.

Stakeholder: I understand, and considering you still have some time left, it should be fine for you to reimplement this system in this newer method that you think will be more efficient.

Stakeholder Signature:



This better method is a PID controller. PID, standing for Proportional-Integral-Derivative, aims to get a current value to a target value, while making sure that it does not get stuck in a loop, oscillating around the target value. This is achieved with the derivative term, which resists the change, slowing down the overshoot that would occur, eventually making sure that the target value would be reached. The only situation where this wouldn't occur would be if the system was experiencing a constant force, such as gravity, in which case it would end up settling around a value that was offset from the target. The integral component will sense this error and slowly accumulate it to push the current value back up to the target, while also controlling how responsive the system is. However, a typical PID controller will only work on a single value, so it is repeated 3 times for each of the respective dimensions with the appropriate values, and then the magnitude of the final vector is clamped to the maxMoveSpeed so that the entity does not move extremely fast.

The most difficult direction for the PID controller is the vertical direction, specifically on the drone, since this will require control of all the components of the system and as such, it will be used to test the system. The special feature of the drone is that it should not be limited by gravity, and so the controller has to fight against gravity, which will need the integral component of the system as mentioned above, which is typically not used by many people implementing one of these controllers.

The main downside to a PID controller is that it needs to be tuned appropriately, of which the values change between different entities. This can take time and as such, I will not be tuning it until it is confirmed that there are no errors with the system.

```
protected virtual float PIDUpdate(float timestep, float currentValue, float targetValue, ref float lastError, ref float storedIntegral){
    float error = targetValue - currentValue; // calculate the difference
    float force = proportionalGain * error; // Calculate the P term
    float derivative = (error - lastError) / timestep; // Rate of change
    derivative *= derivativeGain; // calculate the D term
    storedIntegral = Mathf.Clamp(storedIntegral + error * timestep, -maxStoredIntegral, maxStoredIntegral); // Make sure the stored errors do not exceed
    float integral = integralGain * storedIntegral; // calculate the I term

    lastError = error; // Store the current error value for the next iteration
    return force + integral + derivative; // Return the total
}
```

Above is the standard basic code for a PID controller, where each term is calculated individually and then summed together.

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	The target is outside NearRange	As the most common distance for the target from the player it should work as the valid test	The entity will approach the target and then settle at some point between NearRange and FarRange	N/A	Not applicable without tuning the system
Boundary	The target is inside NearRange	Being very close to the target can cause the system to apply too much force and cause the entity to fly off.	The entity should move away and then return to an appropriate distance quickly	N/A	Not applicable without tuning the system
Boundary	The entity is at the target	The entity reaching the target is a required part of its movement while it is in the Searching state, and as such will need to be tested	The entity will not move	Failed: DivideByZero exception	The change in error being zero is likely causing an error
Erroneous	No target is	This is a test	The entity	Passed	

	set	for a potential error in case the target that the entity is tracking disappears and hence should be handled appropriately	will not move, no exception will be raised		
Erroneous	The target is out of reach of the entity	It is incredibly likely that the player will get somewhere that the entity will be unable to reach, and as such it should be able to handle with this	The entity will attempt to reach as close to the target as possible	N/A	



[14:14:35] DivideByZeroException: Attempted to divide by zero.  
 EnemyEntity.PIDUpdate (System.Single timeStep, System.Single currentValue, System.Single targetValue, System.Single& lastPosition, System.Single& lastError, System.Single& storedIntegral) (a)

When the target is at the same location as the enemy entity, the calculation for the derivative leads to a DivideByZeroException As such this needs to be checked beforehand. I will also take this opportunity to add a different method for calculating the error via the current position of the entity, and to check whether to use one or the other, I will be using an enumerator, as this helps with maintainability.

```
protected virtual float PIDUpdate(float timeStep, float currentValue, float targetValue, ref float lastPosition, ref float lastError, ref float storedIntegral){
    float error = targetValue - currentValue; // Calculate the difference
    float force = proportionalGain * error; // Calculate the P term
    float derivative = 0; // Default derivative value
    if (derivativeType == DerivativeType.Error){ // If using the error to calculate the derivative
        if (error - lastError != 0){ // Verify that an error will not occur
            derivative = (error - lastError) / timeStep; // Rate of change
        }
    } else { // If using the position to calculate the derivative
        if (currentValue - lastPosition != 0){ // Verify that an error will not occur
            derivative = (currentValue - lastPosition) / timeStep; // Rate of change
        }
    }
    derivative *= derivativeGain;
    storedIntegral = Mathf.Clamp(storedIntegral + error * timeStep, -maxStoredIntegral, maxStoredIntegral); // Make sure the stored errors do not exceed the limit
    float integral = integralGain * storedIntegral; // Calculate the I term

    lastError = error; // Store the current error value for the next iteration
    return force + integral + derivative; // Return the total
}
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	The target is outside NearRange	As the most common distance for the target	The entity will approach the target and then settle at	N/A	Not applicable without tuning the

		from the player it should work as the valid test	some point between NearRange and FarRange		system
Boundary	The target is inside NearRange	Being very close to the target can cause the system to apply too much force and cause the entity to fly off.	The entity should move away and then return to an appropriate distance quickly	N/A	Not applicable without tuning the system
Boundary	The entity is at the target	The entity reaching the target is a required part of its movement while it is in the Searching state, and as such will need to be tested	The entity will not move	Passed	
Erroneous	No target is set	This is a test for a potential error in case the target that the entity is tracking disappears and hence should be handled appropriately	The entity will not move, no exception will be raised	Passed	
Erroneous	The target is out of reach of the entity	It is incredibly likely that the player will get somewhere that the entity will be unable to reach, and as such it should be able to handle	The entity will attempt to reach as close to the target as possible	N/A	

		with this		
--	--	-----------	--	--

Interestingly, when the target changes, the derivative term skyrockets, and this is likely to cause an error when the function is linked to the movement of the entity. As such, whenever the target changes, the entity will make sure that it skips over calculating the derivative for that frame.

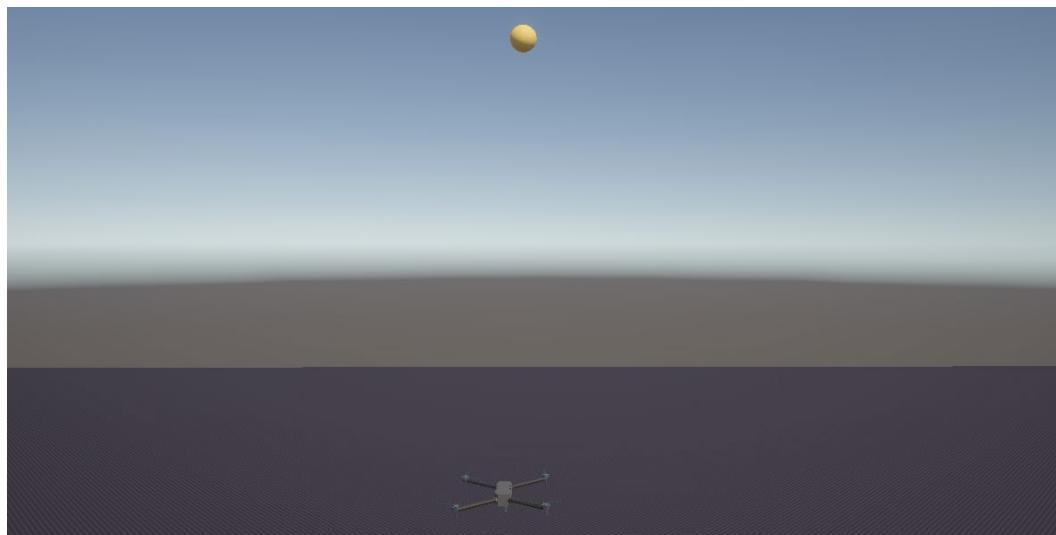
```
protected virtual float PIDUpdate(float timeStep, float currentValue, float targetValue, ref float lastPosition, ref float lastError, ref float storedIntegral){
    float error = targetValue - currentValue; // Calculate the difference
    float force = proportionalGain * error; // Calculate the P term
    float derivative = 0; // Default derivative value
    if (initialised){ // If the target has not recently changed
        if (derivativeType == DerivativeType.Error){ // If using the error to calculate the derivative
            if (error - lastError != 0){ // Verify that an error will not occur
                derivative = (error - lastError) / timeStep; // Rate of change
            }
        } else { // If using the position to calculate the derivative
            if (currentValue - lastPosition != 0){ // Verify that an error will not occur
                derivative = (currentValue - lastPosition) / timeStep; // Rate of change
            }
        }
    } else{ // If the target has recently changed
        initialised = true; // Allows skipping of the error
    }
    derivative *= derivativeGain;
    storedIntegral = Mathf.Clamp(storedIntegral + error * timeStep, -maxStoredIntegral, maxStoredIntegral); // Make sure the stored errors do not exceed the limit
    float integral = integralGain * storedIntegral; // Calculate the I term

    lastError = error; // Store the current error value for the next iteration
    return force + integral + derivative; // Return the total
}
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	The target is outside NearRange	As the most common distance for the target from the player it should work as the valid test	The entity will approach the target and then settle at some point between NearRange and FarRange	N/A	Not applicable without tuning the system
Boundary	The target is inside NearRange	Being very close to the target can cause the system to apply too much force and cause the entity to fly off.	The entity should move away and then return to an appropriate distance quickly	N/A	Not applicable without tuning the system
Boundary	The entity is at the target	The entity reaching the target is a required part of its movement	The entity will not move	Passed	

		while it is in the Searching state, and as such will need to be tested			
Erroneous	No target is set	This is a test for a potential error in case the target that the entity is tracking disappears and hence should be handled appropriately	The entity will not move, no exception will be raised	Passed	
Erroneous	The target is out of reach of the entity	It is incredibly likely that the player will get somewhere that the entity will be unable to reach, and as such it should be able to handle with this	The entity will attempt to reach as close to the target as possible	N/A	

Now that there are no exceptions raised, I can now train the drone correctly. This will be done by using a target and setting its position as the searchingTarget for the controller.



(The drone and the target)

To apply the movement for each direction, the PID function is called on each component, and then passed to a Vector3 to combine it together. This is finally applied to the rigidbody. For debugging purposes, each of the axis of movement can be drawn with a ray, along with the total of all of them.

```
protected override void ApplyMovement(Vector3 target)
{
    float movementX = PIDUpdate(Time.fixedDeltaTime, rb.transform.position.x, target.x, ref lastPosition.x, ref lastError.x, ref storedIntegral.x);
    float movementY = PIDUpdate(Time.fixedDeltaTime, rb.transform.position.y, target.y, ref lastPosition.y, ref lastError.y, ref storedIntegral.y);
    float movementZ = PIDUpdate(Time.fixedDeltaTime, rb.transform.position.z, target.z, ref lastPosition.z, ref lastError.z, ref storedIntegral.z);
    Vector3 movementTotal = new Vector3(movementX, movementY, movementZ); // Combine the movement
    Debug.DrawRay(rb.transform.position, new Vector3(movementX, 0f, 0f).normalized, Color.red); // X direction
    Debug.DrawRay(rb.transform.position, new Vector3(0f, movementY, 0f).normalized, Color.green); // Y direction
    Debug.DrawRay(rb.transform.position, new Vector3(0f, 0f, movementZ).normalized, Color.blue); // Z direction
    Debug.DrawRay(rb.transform.position, movementTotal.normalized, Color.black); // Total direction
    rb.AddForce(movementTotal);
}
```

The proper way to train a controller is by tuning each component in order. First, the proportional term is raised until the controller is considered stable – when the oscillations in its movement very slowly decrease. If they grow, then the controller is considered unstable. Next, the integral term is raised until the controller becomes unstable. Finally, the derivative term is raised to an acceptable level, where the controller still responds quickly but does not overshoot its target. Training a controller takes a long time since changes can take a while to propagate properly.

However, trying to train it is extremely difficult, and it seemed like any changes that were occurring were not causing any effect in how unstable the controller was. Scanning through the code, the problem seems to be caused by the below code.

```
rb.AddForce(movementTotal);
```

By default, Rigidbody.AddForce() uses the Impulse mode, which applies an instantaneous force. This is likely causing an issue to the system, and this needs to be changed to the VelocityChange mode like below.

```
rb.AddForce(movementTotal, ForceMode.VelocityChange);
```

With this change, the controller is much easier to train properly. At this point, I can add on the maximum move speed and the maximum stored integral.

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	The target is outside NearRange	As the most common distance for the target from the player it should work as the valid test	The entity will approach the target and then settle at some point between NearRange and FarRange	Passed	

Boundary	The target is inside NearRange	Being very close to the target can cause the system to apply too much force and cause the entity to fly off.	The entity should move away and then return to an appropriate distance quickly	Passed	
Boundary	The entity is at the target	The entity reaching the target is a required part of its movement while it is in the Searching state, and as such will need to be tested	The entity will not move	Passed	
Erroneous	No target is set	This is a test for a potential error in case the target that the entity is tracking disappears and hence should be handled appropriately	The entity will not move, no exception will be raised	Passed	
Erroneous	The target is out of reach of the entity	It is incredibly likely that the player will get somewhere that the entity will be unable to reach, and as such it should be able to handle with this	The entity will attempt to reach as close to the target as possible	N/A	

Max Move Speed	100
Weapon	Gun
Velocity	X 0 Y 0 Z 0
Proportional Gain	0.1
Derivative Gain	0.05
Integral Gain	0.01
Stored Integral	X 0 Y 19.6 Z 0
Max Stored Integral	50
Derivative Type	Error

(The values for the drone)

The next function that can be added is the ability for the entity to select its next target. This can be produced by randomly selecting a point in a sphere around the player, which can be made by randomly generating a float for each axis.

```
protected void SelectTarget(){
    searchingTarget.x = Random.Range(-DetectionRange, DetectionRange); // Generate X point
    searchingTarget.y = Random.Range(-DetectionRange, DetectionRange); // Generate Y point
    searchingTarget.z = Random.Range(-DetectionRange, DetectionRange); // Generate Z point
    searchingTarget = searchingTarget + rb.transform.position; // Make the point relative to the current position of the player
    ResetInitialisation(); // Reset the derivative
}
```

Testing whether this function works correctly or not is based upon whether the entity behaves properly afterwards. Witnessing the drone move, 2 issues became obvious:

- Firstly, the drone would continue moving upwards. While it may also move downwards, it would still move too far from the ground, or from the nearest piece of level geometry.
- The selected target is sometimes not accessible to the entity, since it is behind an object.

To fix the first problem, the entity needs to check where the closest point is. Unity does have a built-in function to work out the distance between the closest point on 2 colliders, and this can be combined with Physics.OverlapSphere() to find all the colliders in a certain range from the entity.

```
protected Vector3 GetClosestPoint(out bool found){
    Collider[] colliders = Physics.OverlapSphere(
        position: rb.transform.position,
        radius: DetectionRange * 2
    ); // Get list of colliders nearby
    found = false;
    Vector3 point = Vector3.zero;
    float distSqrD = Mathf.Infinity; // Start off with an infinite distance
    foreach (Collider collider in colliders){
        found = true;
        if ((rb.transform.position - collider.transform.position).sqrMagnitude < distSqrD){ // if the distance to this collider is less than the current smallest
            distSqrD = (rb.transform.position - collider.transform.position).sqrMagnitude; // Set its distance as the new smallest one
            point = collider.ClosestPoint(rb.transform.position); // Get the exact distance to the point
        }
    }
    return point;
}
```

Using the square of the distance is a minor optimisation since the square root is an expensive thing to calculate and hence minimising the number of times I need to calculate it, the faster the game is. This is fine to do since the usage of distSqr is only for comparing with other distances. Since the square of 1 distance will always be larger than another distance that is smaller, nothing breaks from doing this.

This function can then be incorporated into SelectTarget() like below:

```
bool found;
Vector3 point = GetClosestPoint(out found);
if (!found){ // If piece of level geometry is not found within the entities detection range
    searchingTarget.y = rb.transform.position.y - DetectionRange; // Set the target to be below it.
    return;
}
Vector3 entityToPoint = rb.transform.position - point; // Calculate the distance to the found point
if (entityToPoint.magnitude > DetectionRange){ // If the point is outside the range of the entity
    searchingTarget = point - entityToPoint.normalized * Random.Range(0, DetectionRange); // Set the target distance as a random distance towards the closest point.
    return;
}
```

To solve the second problem, a raycast is required to check whether the drone can make it to the target from the current position, and if not, the target just needs to be rechecked. I will also take this opportunity to add a check in to make sure that the target is not too close to the entity.

```
bool accessible = false;
while (!accessible){ // While the target is not accessible, generate a new point
    searchingTarget.x = Random.Range(-DetectionRange, DetectionRange); // Generate X point
    searchingTarget.y = Random.Range(-DetectionRange, DetectionRange); // Generate Y point
    searchingTarget.z = Random.Range(-DetectionRange, DetectionRange); // Generate Z point
    searchingTarget = searchingTarget + rb.transform.position; // Make the point relative to the current position of the player
    RaycastHit hitInfo;
    if (Physics.Raycast( // Check that there is no object between the target and the entity
        origin: rb.transform.position,
        direction: (searchingTarget - rb.transform.position).normalized,
        hitInfo: out hitInfo,
        maxDistance: DetectionRange
    )){
        if ((hitInfo.point - rb.transform.position).magnitude < 1f){ // If the point is too close to the entity
            continue;
        } else { // The point is set
            searchingTarget = hitInfo.point;
            accessible = true; // Used to exit the loop
        }
    }
}
ResetInitialisation(); // Reset the derivative
```

Retesting the search target again leads to no visual errors in the entity's behaviours, which means the next thing to add was the ability to search for the player. Searching is typically done via firing multiple raycasts out in the direction that you want to search. These are also typically shot out in an arc to cover a wider range. Furthermore, since the rays get shot out quite far, the distance between the ends of 2 neighbouring rays can be far more than the width of the player, and as such more rays need to be fired, and the angle between the rays needs to be decreased. To calculate the angle, I need to divide the viewing angle, which is 90 degrees, with the length of the arc formed by the 2 furthest rays. This arc length can be rounded to the next integer to work out how many rays are required. These values are then calculated once when the entity is initialised.

```
public void SetAngles(){ // Finds the angle needed to make sure that the player is not missed between rays
    float quarterCircumference = 2 * Mathf.PI * DetectionRange * 0.25f; // Find the arc length
    rayCount = Mathf.CeilToInt(quarterCircumference); // Calculate the number of rays
    rayAngle = 90 / quarterCircumference; // Calculate the angle between each ray
}
```

Firing each ray can be done sequentially and if any of these rays connect with a target, then the target is set, and the entity enters combat mode.

```
protected void Searching(){
    for (int i = 0; i < rayCount; i++)
    {
        RaycastHit hitInfo;
        Vector3 rayDirection = Quaternion.AngleAxis(-45f + rayAngle * i, Vector3.up) * Vector3.forward; // Calculate ray direction
        rayDirection = rb.transform.rotation * rayDirection; // Make it relative to the entity
        Debug.DrawRay(rb.transform.position, rayDirection, Color.red);
        if (Physics.Raycast( // Check if a ray has intersected with a target
            origin: rb.transform.position,
            direction: rayDirection,
            hitInfo: out hitInfo,
            maxDistance: DetectionRange,
            layerMask: playerLayer
        )){
            Debug.DrawRay(rb.transform.position, rayDirection, Color.green);
            state = EnemyManager.MemberState.Combat; // Set state to combat
            target = hitInfo.collider.gameObject; // Set target to the player's gameobject
            break;
        }
    }
}
```

The entity also needs to rotate towards its target to make sure that it is searching for its location properly. This can be done by finding the direction towards the target and converting this into a rotation. Finally, the current rotation and the intended one can be interpolated between to add some leniency for the player. This makes it possible for the player to get around the entity and avoid getting hit.

```
protected override void RotateToTarget(Transform target){
    Vector3 targetDirection = (target.transform.position - rb.transform.position).normalized;
    Quaternion dirInQuaternion = Quaternion.LookRotation(targetDirection);
    rb.transform.rotation = Quaternion.Slerp(rb.transform.rotation, dirInQuaternion, RotationSpeed * Time.fixedDeltaTime);
}
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	The player is in front of the entity	This is the most typical way that the player will be detected by the entity will detect the player	The entity will enter combat mode	Passed	
Valid	The player is not in front of entity	The drone will not always see the player and as such must search as usual	The entity will continue searching and moving towards the target	Passed	
Boundary	The player is	The player	The entity will	Passed	

	on the edge of the entity's view	might be moving quickly and only be in the drone's view for a moment and as such this must be tested appropriately	enter combat mode		
Boundary	The player is on the edge of the entity's detection range	The player may attempt to escape the drone by being on the edge of its view, and hence it should still detect the player	The entity will enter combat mode	Passed	

The next state of the entity that needs to be handled is the combat state. In this, the movement target needs to be set some distance away from the player. This distance will be the midpoint between the FarRange and NearRange. When the entity's weapon is able to fire at the player, the entity will call Fire() on the weapon to deal damage to it.

```
protected virtual void Combat(){ // Fire weapon
    RaycastHit hitInfo;
    if (Physics.Raycast( // Check if there the player is in the weapon's view
        origin: rb.transform.position,
        direction: weapon.transform.GetChild(0).forward,
        hitInfo: out hitInfo,
        maxDistance: DetectionRange
   )){
        GameObject hit = hitInfo.collider.gameObject; // Grab the hit gameobject
        Debug.Log($"Enemy {gameObject.GetInstanceID()} facing {hit.GetInstanceID()}");
        Entity entityClass = hit.GetComponent<Entity>(); // Get the entity class from the object
        if (entityClass != null){ // If the object is damageable
            weaponScript.Fire(); // Fire the weapon
            Debug.Log($"Enemy {gameObject.GetInstanceID()} fired its {weaponScript.WeaponType}");
        }
    }
}
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	The player is in front of the entity	This is the most typical way that the	The entity will fire its weapon and	Passed	

	while it is in combat mode	player will be detected by the entity will detect the player	move away		
Valid	The player is not in front of entity while it is in combat mode	The drone will not always see the player and as such must rotate towards them	The entity will rotate towards the player	Passed	
Boundary	The player is on the edge of the entity's view while it is in combat mode	The entity also needs to be able to conserve ammunition and this makes sure that they do not fire at all moments	The entity will rotate towards the player	Passed	
Boundary	The player is on the edge of the entity's detection range while it is in combat mode	The player may attempt to escape the drone by being on the edge of its view, and hence it should still detect the player	The entity will chase after the player and fire if possible	Passed	

## End of prototype tests

### *Entity Movement*

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	The target is outside NearRange	As the most common distance for the target from the player it should work as the valid	The entity will approach the target and then settle at some point between NearRange and		

		test	FarRange		
Boundary	The target is inside NearRange	Being very close to the target can cause the system to apply too much force and cause the entity to fly off.	The entity should move away and then return to an appropriate distance quickly		
Boundary	The entity is at the target	The entity reaching the target is a required part of its movement while it is in the Searching state, and as such will need to be tested	The entity will not move		
Erroneous	No target is set	This is a test for a potential error in case the target that the entity is tracking disappears and hence should be handled appropriately	The entity will not move		
Erroneous	The target is out of reach of the entity	It is incredibly likely that the player will get somewhere that the entity will be unable to reach, and as such it should be able to handle with this	The entity will attempt to reach as close to the target as possible		

*Entity Searching*

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	The player is in front of the entity	This is the most typical way that the player will be detected by the entity will detect the player	The entity will enter combat mode		
Valid	The player is not in front of entity	The drone will not always see the player and as such must search as usual	The entity will continue searching and moving towards the target		
Boundary	The player is on the edge of the entity's view	The player might be moving quickly and only be in the drone's view for a moment and as such this must be tested appropriately	The entity will enter combat mode		
Boundary	The player is on the edge of the entity's detection range	The player may attempt to escape the drone by being on the edge of its view, and hence it should still detect the player	The entity will enter combat mode		

*Entity Combat*

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	The player is in front of the entity	This is the most typical way that the	The entity will fire its weapon and		

	while it is in combat mode	player will be detected by the entity will detect the player	move away		
Valid	The player is not in front of entity while it is in combat mode	The drone will not always see the player and as such must rotate towards them	The entity will rotate towards the player		
Boundary	The player is on the edge of the entity's view while it is in combat mode	The entity also needs to be able to conserve ammunition and this makes sure that they do not fire at all moments	The entity will rotate towards the player		
Boundary	The player is on the edge of the entity's detection range while it is in combat mode	The player may attempt to escape the drone by being on the edge of its view, and hence it should still detect the player	The entity will chase after the player and fire if possible		

## Interview

**Stakeholder:** Upon trying out the new system, I think it is a little monotonous and frustrating for the player to have to reload every time they want to fire the weapon. The weapon should instead automatically reload if the player attempts to fire it when there is no ammunition remaining. Furthermore, I've realised that having to toggle the zoom action is monotonous, and as such it should be enabled while the player is holding down the button. Finally, it is a little difficult to tell what the player is firing at, so the current weapon needs to point towards the target.

Stakeholder Signature:



## Prototype 3

### Automatically Reloading

To fulfil the stakeholder's request of making the weapon automatically reload, the gun needs to check if there is no ammo remaining when the player requests it to fire, and if so, it needs to reload the weapon.

```
override public void Fire(){ // Instantiates a bullet at the gun's barrel traveling outwards
    if (AmmoInClip > 0){
        Instantiate(bullet, barrel.transform.position, barrel.rotation);
        Debug.Log("Fired " + gameObject.name);
        TotalAmmo -= 1;
        AmmoInClip -= 1;
    } else { // Auto reload the weapon if there isn't any ammo Left
        Reload();
    }
}
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the fire button while there is enough ammunition remaining	This is the most common action the weapon will encounter and therefore is suitable as a valid test	The gun will fire 1 bullet	Passed	
Boundary	Pressing the fire button while there is no ammunition remaining	It is important the system still acts accordingly when there is no ammunition	The gun will reload	Passed	

		in the clip remaining			
Erroneous	Pressing another key (e.g. M)	Any inputs not related to the system should not affect it	The gun will not fire	Passed	

Stakeholder: Yes, this is much better. This way the player will be able to play more fluidly and easier. You can move onto the next system.

Stakeholder Signature:



## Held Zoom

To fulfil the stakeholder's request of making the zoom action not toggled, but instead held, the check needs to be changed to checking whether the button is being currently held. The function then needs to be run every frame within the Update() loop, instead of being activated on the input being detected.

```
private void Zoom(){
    if (playerInputs.Player.Zoom.inProgress){
        playerHead.fieldOfView = zoomFov; // Set the camera's fov to the zoom fov
        lookScript.sensitivity = lookScript.zoomSensitivity; // Lower the sensitivity
    } else {
        playerHead.fieldOfView = fov; // Reset the camera's fov
        lookScript.sensitivity = lookScript.normalSensitivity; // Reset the sensitivity
    }
}
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the fire button while there is enough ammunition	This is the most common action the weapon will	The gun will fire 1 bullet	Passed	

	remaining	encounter and therefore is suitable as a valid test			
Boundary	Pressing the fire button while there is no ammunition remaining	It is important the system still acts accordingly when there is no ammunition in the clip remaining	The gun will reload	Passed	
Erroneous	Pressing another key (e.g. M)	Any inputs not related to the system should not affect it	The gun will not fire	Passed	

Stakeholder: Yes, this makes it a lot faster to change from zooming in to back out, which is a lot nicer to play as instead.

Stakeholder Signature:



### Weapon point to target

To make it more obvious what the player will hit when firing their weapon, the weapon will point towards the point that they will hit. To make it look better and not just snapping, the weapon will rotate towards the target quickly but smoothly. This can be done with Quaternion.Slerp() which spherically interpolates between 2 vectors, which are the current facing direction and the vector from the gun to the aiming location.

```

private void Update() {
    Zoom(); // Check the fov state
    if (activeWeaponSlot >= maxWeapons || activeWeaponSlot < 0){ // if the current weapon slot is outside the maximum weapon range
        activeWeaponSlot = Wrap(activeWeaponSlot, weaponSlots); // Wrap it appropriately
    }
    if (weaponSlots.Count > 0){ // If there is a weapon active
        RaycastHit playerHit;
        Quaternion lookRotation = playerHead.transform.rotation; // temporarily store the current facing rotation
        if (Physics.Raycast( // if this raycast hits something
            origin: playerHead.transform.position,
            direction: playerHead.transform.rotation * Vector3.forward,
            hitInfo: out playerHit
        )){
            Vector3 pointDirection = (playerHit.point - weaponSlots[activeWeaponSlot].transform.position).normalized; // find the direction the weapon needs to point
            towards
            lookRotation = Quaternion.LookRotation(pointDirection); // Store the required rotation to reach the target
        }
        weaponSlots[activeWeaponSlot].transform.rotation = Quaternion.Slerp(weaponSlots[activeWeaponSlot].transform.rotation, lookRotation, rotationSpeed * Time.deltaTime); // smoothly turn the weapon towards the
    }
}

```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Looking from a close object to a far object or vice versa	While purely cosmetic, it will provide helpful information to the player on where they are pointing	The weapon will point closer to the body of the player and then further away, pointing at the object at the centre of the screen	Passed	
Valid	Not moving the mouse after moving it	The function should finish its last action	The weapon will finish its rotation.	Passed	
Valid	Not moving the mouse	Since the input data is the same, the output should also be the same	The weapon will not rotate	Passed	
Boundary	Looking from a close object to a far object or vice versa while the player is moving quickly	The function should work independent of the player's movement, and the physics engine of Unity can get a dodgy at high speeds	The weapon will point closer to the body of the player and then further away, pointing at the object at the centre of the screen	Failed: the weapon kept pointing towards the player	The physics system seems to be returning an incorrect result; therefore, the player should be masked out

Erroneous	Pressing another key (e.g. M)	Any inputs not related to the system should not affect it	The weapon will not rotate	Passed	
-----------	-------------------------------	---	----------------------------	--------	--

At high speeds, it seems that the physics system of Unity is detecting that the object that the player is pointing at the player, and to fix this the weapon needs to exclude the player pointing at themselves, and this can be done with a layer mask. Within Physics.Raycast(), the optional layerMask argument specifies what the raycast should include in the results, contrary to the typical use of a LayerMask object. Coincidentally, Unity does not have an overload of Physics.Raycast() that only takes an origin, direction, LayerMask, and returns data about its output to a referenced variable. The closest function that contains all these parameters also includes a maximum distance field, but this can be made irrelevant by passing in infinity as the maximum distance like below:

```
if (Physics.Raycast( // if this raycast hits something
    origin: playerHead.transform.position,
    direction: playerHead.transform.rotation * Vector3.forward,
    hitInfo: out playerHit,
    layerMask: notPlayerLayer,
    maxDistance: Mathf.Infinity
)){
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Boundary	Looking from a close object to a far object or vice versa while the player is moving quickly	The function should work independent of the player's movement, and the physics engine of Unity can get a dodgy at high speeds	The weapon will point closer to the body of the player and then further away, pointing at the object at the centre of the screen	Passed	

**Stakeholder:** While subtle, this does add a much-needed touch to the weapon's behaviour. You may move onto the next system, since this one has been finalised.

Stakeholder Signature:



## Saving and Loading System

To begin the save system, there first needs to be data that can be stored. All of this save data will be stored in a file that is formatted as a JSON, which is a very useful system to describe and store the contents of an object in a text-based format. While there are other formats such as the XML format, Unity has built in support for the JSON format with the Newtonsoft.Json library that is built into the engine. This allows both the serialization class and the deserialization of a file back into an object instance.

The data that the player would want to store is everything about their current state. This can be split into different sections, such as data related to the player, data related to story events, and data related to the level highscores. The player specific data includes:

- The level that they are in
- Their health
- Their maximum health
- Their position
- Their velocity
- Their rotation (in both looking directions)
- Their currently equipped weapons
- The currently selected weapon
- The amount of time the player has played in that save

The data that is related to story events is completely arbitrary, and hence only the underlying system needs to be in place to save it. Accessing this data will come later in development and the same goes for the level highscores. These 3 types of data are kept as their own classes, which are all instantiated into another class called SaveData. This is the class that is serialized into the file and is a one-stop location for all the information that anything in the system needs.

```

using UnityEngine;

[System.Serializable]
public class SaveData
{
    public PlayerData playerData;
    public WorldFlags worldFlags;
    public LevelHighscores levelHighscores;

    public SaveData()
    {
        playerData = new PlayerData();
        worldFlags = new WorldFlags();
        levelHighscores = new LevelHighscores();
    }
}

```

Each of these classes will have constructors that initialise them with default data, which will be used when the player wants to make a new save. However, for the player's position and velocity, along with their rotation, things get a bit more difficult. This is because the built-in Vector3 and Quaternion classes are not marked as serializable and hence cannot be written to a file. Therefore, I need to make versions of them that can be serialized, and this is done by taking the only the important components and copying them into an appropriate structure. Extra operations need to be added for the implicit casting operators, which helps to improve maintainability by making them as easy to use.

```

[System.Serializable]
public struct SerVector3
{
    public float x;
    public float y;
    public float z;

    public SerVector3(float x, float y, float z)
    {
        this.x = x;
        this.y = y;
        this.z = z;
    }

    public static implicit operator Vector3(SerVector3 v)
    { // This is used for casting from SerVector to Vector3
        return new Vector3(v.x, v.y, v.z);
    }

    public static implicit operator SerVector3(Vector3 v)
    { // This is used for casting from Vector3 to SerVector
        return new SerVector3(v.x, v.y, v.z);
    }
}

[System.Serializable]
public struct SerQuaternion
{
    public float x;
    public float y;
    public float z;
    public float w;

    public SerQuaternion(float x, float y, float z, float w)
    {
        this.x = x;
        this.y = y;
        this.z = z;
        this.w = w;
    }

    public static implicit operator Quaternion(SerQuaternion q)
    { // This is used for casting from SerQuaternion to Quaternion
        return new Quaternion(q.x, q.y, q.z, q.w);
    }

    public static implicit operator SerQuaternion(Quaternion q)
    { // This is used for casting from Quaternion to SerQuaternion
        return new SerQuaternion(q.x, q.y, q.z, q.w);
    }
}

```

These are then used within PlayerData.

```
[System.Serializable]
public class PlayerData
{
    public string currentScene;
    public float health;
    public float maxHealth;
    public SerVector3 position;
    public SerVector3 velocity;
    public SerQuaternion bodyRotation;
    public SerQuaternion lookRotation;
    public List<WeaponData> weaponSlots;
    public int activeWeaponSlot;
    public int merit;
    public float sanity;
    public float timeOnSave;
    public int totalKills;

    public PlayerData()
    { // Initialise PlayerData with some default values
        currentScene = "Tutorial";
        health = 100f;
        maxHealth = health;
        position = new SerVector3(0, 0.85f, 0);
        velocity = new SerVector3(0, 0, 0);
        bodyRotation = new SerQuaternion(0, 0, 0, 0);
        lookRotation = new SerQuaternion(0, 0, 0, 0);
        weaponSlots = new List<WeaponData>();
        activeWeaponSlot = 0;
        merit = 0;
        sanity = 100f;
        timeOnSave = 0;
        totalKills = 0;
    }
}
```

Since there will only be a maximum of 4 weapons, weaponSlots is initialised as an array to keep the memory usage as low as possible. To update the save file, the game manager needs to pass through a reference to the player to the data, at which point the system will read the relevant values it needs.

```
public void UpdateSaveData(GameObject player)
{
    // Update the player's save data
    playerData.UpdateData(player);
}
```

These values span from different parts of the game, although the majority are from scripts that are attached to the player's GameObject as components. These can be retrieved, and then the public variables can be accessed properly.

```
public void UpdateData(GameObject player)
{
    currentScene = SceneManager.GetActiveScene().name; // Get the scene the player is in
    health = player.GetComponent<PlayerEntity>().Health; // Get the player's health from their entity script
    maxHealth = player.GetComponent<PlayerEntity>().MaxHealth; // Get the player's maximum health from their entity script
    position = player.transform.position; // Get the player's position
    velocity = player.GetComponent<Rigidbody>().linearVelocity; // Get the player's current velocity from the rigidbody attached to it
    bodyRotation = player.transform.rotation; // Get the player's current rotation
    lookRotation = Camera.main.transform.rotation; // Get the camera's rotation
    weaponSlots = new List<WeaponData>(); // Clear the weapon slots
    foreach (GameObject weapon in player.GetComponent<PlayerGunInteraction>().weaponSlots) // Get a list every weapon currently equipped
    {
        Weapon weaponScript = weapon.GetComponent<Weapon>(); // Get the underlying Weapon script of the weapon
        weaponSlots.Add(new WeaponData(weaponScript.WeaponType, weaponScript.TotalAmmo, weaponScript.AmmoInClip, weaponScript.ClipCapacity)); // Make new
        // weapon save data and add it to the list
    }
    activeWeaponSlot = player.GetComponent<PlayerGunInteraction>().activeWeaponSlot; // Save the position of the currently equipped weapon
    merit = PlayerManager.Instance.merit; // Assign merit
    sanity = PlayerManager.Instance.sanity; // Assign sanity
    timeOnSave += Time.realtimeSinceStartup; // Add on the time that the player has had the game open
    totalKills = PlayerManager.Instance.totalKills; // Get the total kills the player has done
}
```

However, handling weapons are different, since the GameObject class is not a serializable one, and if it was, it would cause a lot of unnecessary data to be stored in the player's save file. As such, only the information stored in the abstract Weapon class is written down, in a second temporary format which only has these properties. Since this is marked as serializable, this can be added to the file, and then a separate system can work out what prefabricated object (prefab for short) needs to be loaded into the player's weapon slots.

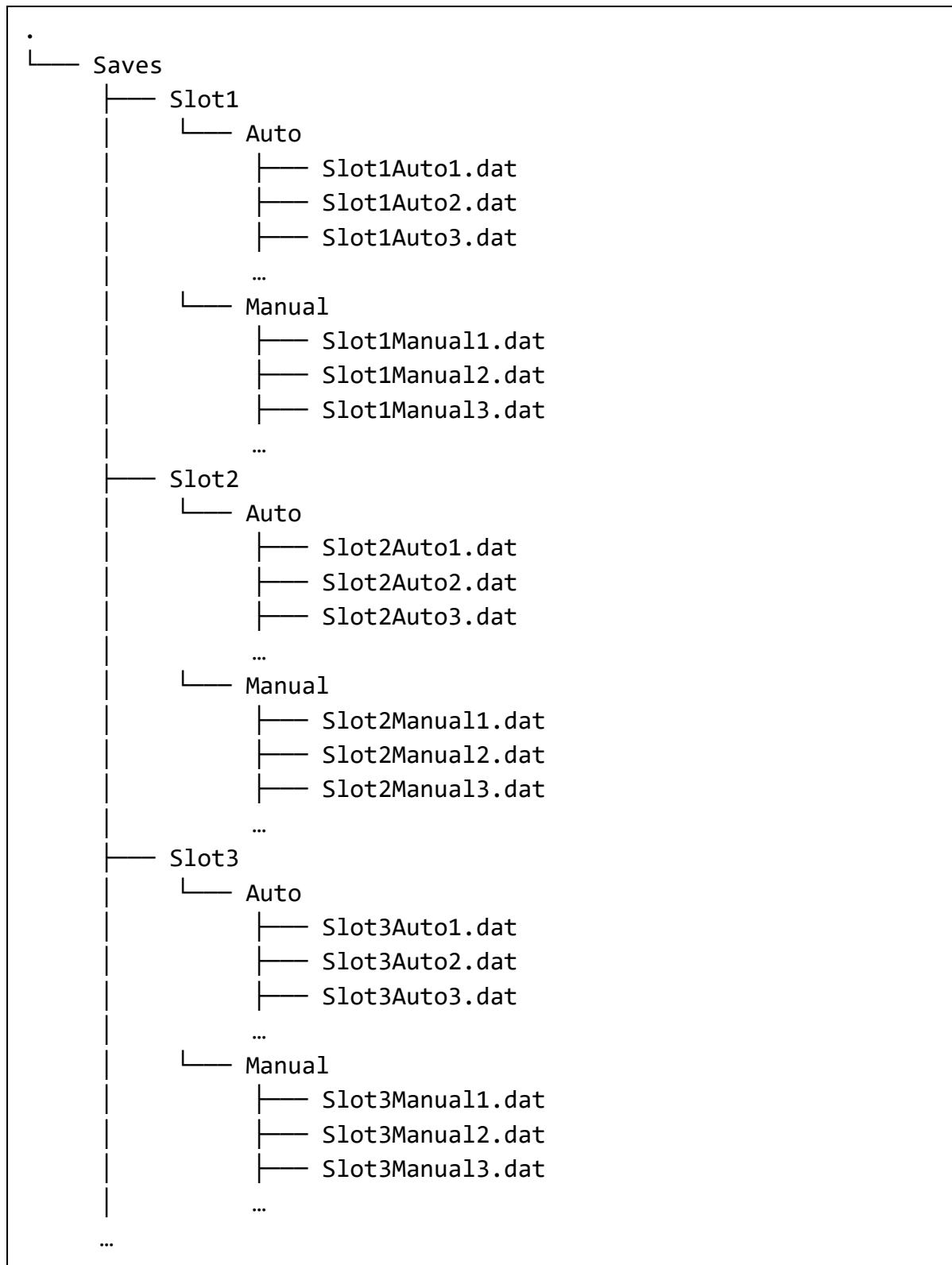
```
[System.Serializable]
public class WeaponData
{
    public string name;
    public int totalAmmo;
    public int ammoInClip;
    public int clipCapacity;
    public WeaponData(string name, int totalAmmo, int ammoInClip, int clipCapacity)
    { // Copies the important weapon data into a smaller format
        this.name = name;
        this.totalAmmo = totalAmmo;
        this.ammoInClip = ammoInClip;
        this.clipCapacity = clipCapacity;
    }
}
```

Now that the data has been stored in a serializable class, this class next needs to be turned into JSON, and then written as a file. All saves have to be stored in a location where the saves will not be deleted or moved by the system, and Unity has a built-in location for this, referenced by Application.persistentDataPath. Using the Path library, I can create a folder path that will contain the saves in the right place.

There are 2 types of saves:

- Auto – these are automatic ones done by the game every time a level ends, or when a checkpoint is reached.
- Manual – these are made whenever the player manually clicks save in the pause menu.

Furthermore, the player will have access to as many slots as they want, their own storage providing. As such, each save needs to reference the slot it is in, along with the type of save it is, allowing for a much easier experience whenever the saves needed to be accessed manually through the folder structure, which would look like below:



This repeatable pattern can be used to sort the files in one of the directories, resulting in the most recent save being last in the list. Using regular expressions, I can retrieve the save number, at which point I can increment it to finish making the name for the new save. The next step is to write the file and save it. This can be done with a `StreamWriter`, which will also create the file in the process. The prepared `SaveData` is now serialized

using JsonConvert.SerializeObject(), and to help with maintainability, the JSON is indented. At this point, the player's data has been saved, but the function is not yet over. When the player needs to select a game slot, the game needs to know which save file the player last used, and hence, there needs to be a file that stores this information. This file will be stored in the same directory as the top-level 'Saves' folder.

```
public static void Save(SaveData data, string type, int saveSlot)
{
    type = new CultureInfo("en-US", false).TextInfo.ToTitleCase(type); // Make sure the type is following the naming convention
    string path = Path.Combine(Application.persistentDataPath, "Saves", "Slot" + saveSlot.ToString() + type); // Make the path to the folder
    string fileMatch = "Slot" + saveSlot.ToString() + type + "*.dat"; // Get the pattern for the important files in the folder
    string[] matches = Directory.GetFiles(path, fileMatch); // Get all files that match the pattern
    Array.Sort(matches, new CompareSaves()); // Sort the files based on the last number
    int fileNum = 1; // Default value if no files are found
    if (matches.Length > 0)
    {
        string lastFile = matches[0]; // Get the latest file in the directory
        lastFile = Path.GetFileNameWithoutExtension(lastFile).ToString(); // Get only the name of the file
        fileNum = Int32.Parse(Regex.Match(lastFile, @"\d+").ToString()) + 1; // Get the number of the file and increment it
    }
    path = Path.Combine(path, "Slot" + saveSlot.ToString() + type + fileNum.ToString() + ".dat"); // Make the name of the new file

    using (StreamWriter stream = new StreamWriter(path)) // Make the file on the system
    {
        string json = JsonConvert.SerializeObject(data, Formatting.Indented); // Convert the save data into JSON
        stream.Write(json); // Fill with constructed save data
    }
    // Modify Lookup
    Dictionary<string, string> lookup = GetLookup(); // Read the lookup
    lookup["Slot" + saveSlot] = path; // Edit this slot's new path
    List<string> keys = new List<string>(lookup.Keys); // Get all the keys
    List<string> values = new List<string>(lookup.Values); // Get all the values

    using (StreamWriter lookupWriter = new StreamWriter(Path.Combine(Application.persistentDataPath, "SaveLookup.txt")))
    { // Write over SaveLookup.txt
        for (int i = 0; i < keys.Count; i++) // For each key in the dictionary
        {
            lookupWriter.WriteLine(keys[i] + "," + values[i]); // Write each line with a comma as the separator
        }
    }
}
```

The internals of SaveLookup.txt is a representation of the data stored within a dictionary, of which this file is parsed in and out of when the code needs to use it. This keeps the file size at a minimum, and as such, lowers the requirements of the game. When accessing this file, each line must be separated into the key and value. This separator is a comma, since it is one of the few separator characters that doesn't show up in the full line (folder names may have spaces/underscores/dashes, and the drive letter will have a colon in it).

```
public static Dictionary<string, string> GetLookup()
{
    if (!File.Exists(Path.Combine(Application.persistentDataPath, "SaveLookup.txt")))
    { // If the file doesn't exist
        File.Create(Path.Combine(Application.persistentDataPath, "SaveLookup.txt")).Close(); // Create it and close it
    }

    string[] pathLookup = File.ReadAllLines(Path.Combine(Application.persistentDataPath, "SaveLookup.txt")); // Read every line and store as an array
    Dictionary<string, string> lookupPairs = new Dictionary<string, string>(); // Make a new dictionary
    foreach (string lookup in pathLookup)
    { // For every line in the file
        string[] pairArray = lookup.Split(','); // Split it into pairs
        lookupPairs.Add(pairArray[0], pairArray[1]); // Add each pair into the dictionary
    }
    return lookupPairs; // Return the dictionary
}
```

Once the dictionary has been formed, the relevant slot has the path to the file edited, and then SaveLookup.txt is rewritten.

The opposite action to saving is loading the data and is far simpler in comparison. Loading a file does not need to edit the lookup, nor does it need to sort and search through directories. This leads to the internals of Load() being relatively simple, containing a StreamReader to open the file, and a call for JsonConvert.DeserializeObject() to convert back into SaveData.

```
public static SaveData LoadSave(string path)
{
    SaveData data; // Define the save data
    using (StreamReader sr = new StreamReader(path))
    { // Open the file for reading
        data = JsonConvert.DeserializeObject<SaveData>(sr.ReadToEnd()); // Convert the file back into a SaveData class
    }

    return data; // Return the read data
}
```

To call all of this code, elements of the game will call a representative function in the game manager which will handle any inputs or assignments. The first of these functions is SaveGame(), which first updates the save data, and then requests that the system saves it. Further functions will be added later.

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Saving a new save with others already being made	The most typical action the player will be taking is this, and therefore it should be tested properly	A new save will be created with the newest number	Partial: works fine until the 10 <sup>th</sup> save is made, at which point it is overwritten every time	Strings are sorted differently that integers, and hence needs to be changed
Valid	Loading a save file	It goes hand in hand with saving, and will also be a core function of the system	The save file is opened and the data is returned	Passed	
Boundary	Saving a file when none of the directories or files exist	This is necessary to test for since this will be the state of the user's system when they first launch the game	All the appropriate files and folders are created	Passed	

Error	Passing a file to load that doesn't exist	In case the lookup ends up corrupted, or the user deletes certain files, the function should not crash, and instead log the error	Error code is logged	Failed: No code to pass it through to the log	Needs a check to see if the file exists first
Error	'SaveLookup.txt' is not created	This will occur when the player opens the game for the first time, and must be handled properly	The file is created	Passed	

When strings are compared, they are compared alphabetically. Since this doesn't include numbers, the system also sorts it according to a per-digit alphabetic system, where 1 comes before 10, which comes before 2. This means that when the files are sorted, the last file that comes is number 9, even though 10 already exists. When the file is made, it overwrites 10, as it has the same name. To combat this, `Array.Sort()` does allow a custom comparison function to be passed through. This comparer must be of a certain type, and therefore a small class must be implemented to handle this – `CompareSaves`. This inherits from `IComparer<T>`, where `T` is the specific type `string` (leading to the inherited class being `IComparer<string>`). The only requirement of the parent class is that the `Compare` method is given an implementation, where it takes in 2 arguments, which are the 2 elements being compared. The function must return an integer, this being either 1, 0, or -1, for when the first element is greater, the same as the second, or smaller than the second element respectively. To compare them, I can use the same method that was used to get the number of the file ending earlier – regular expressions. The particular expression that I am using is '`\d+$`', which selects consecutive digits as 1 string, while reading backwards from the end of the string to improve efficiency.

```
public int Compare(string s1, string s2){
    return int.Parse(Regex.Match(Path.GetFileNameWithoutExtension(s1).ToString(), @"\d+$").ToString())
    >
    int.Parse(Regex.Match(Path.GetFileNameWithoutExtension(s2).ToString(), @"\d+$").ToString()) ? 1 : -1;
    // if the number on the first string is larger than the second, return 1, else return -1
}
```

This can then be used on both of the inputs and then compare to find out which one is larger. There is no need to check if they are the same, as the resulting output is incremented anyway.

```
Array.Sort(matches, new CompareSaves());
```

To solve the second problem, I only need to check if the file exists beforehand, and to log an error if it doesn't.

```
if (!File.Exists(path)){
    Debug.Log($"File at '{path}' does not exist.");
}
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Saving a new save with others already being made	The most typical action the player will be taking is this, and therefore it should be tested properly	A new save will be created with the newest number	Passed	
Error	Passing a file to load that doesn't exist	In case the lookup ends up corrupted, or the user deletes certain files, the function should not crash, and instead log the error	Error code is logged	Passed	

The next thing to add is the ability to list out all of the saves, so that when I come to build the GUI, it can leverage this functionality. Listing out the saves requires reading the lookup file and creating a temporary container, which I shall call SaveSlot, which contains the important information about the file. Importantly, it must also check that the top-level 'Saves' folder exists, since on the player's first time opening the game, the folder will not exist. If the folder does not exist, then it needs to create that folder, and then return an empty list, just like if the lookup is empty. If the folder does exist, then every folder within it, which will be all of the slots, must be taken into a list, where it is then used to get the path to the most recent save in that slot. This is then used to create a new SaveSlot, which will contain:

- The slot's number
- The path to the save file
- The current scene name
- The time spent in the save

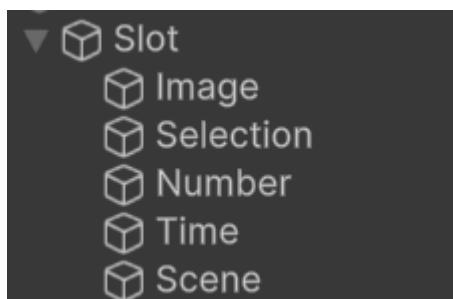
This is all the important information that will be shown in the GUI, as is according to the design of that screen. To retrieve this information, each slot will load their respective saves and access the data and then close it. This is why SaveSlot stores the path and not the actual SaveData, since doing it like this means that game minimises the amount of SaveData open, which could become extremely large as SillyAustralian fills the game with more events and levels. Once this is all completed, the finalised list is then returned at the end.

```
public static List<SaveSlot> ListAllSlots()
{
    Dictionary<string, string> lookupPairs = GetLookup(); // Get the slot pairs
    if (lookupPairs == null) // If there are no saves made
    {
        return new List<SaveSlot>(); // Return an empty list
    }
    string path = Path.Combine(Application.persistentDataPath, "Saves"); // Path to the saves folder
    if (!Directory.Exists(path)) // If no saves have been made
    {
        Directory.CreateDirectory(path); // Create the folder
        return new List<SaveSlot>(); // Return an empty list
    }
    string[] directories = Directory.GetDirectories(path); // Get all the folders within the Saves folder
    List<SaveSlot> slotList = new List<SaveSlot>(); // Create a new
    foreach (string directory in directories)
    { // For every save slot
        string folder = directory.Substring(path.Length).TrimStart(Path.DirectorySeparatorChar); // Get the folder name
        string savePath = lookupPairs[folder]; // Get the filepath
        SaveSlot slot = new SaveSlot(savePath); // Create a new container
        slotList.Add(slot); // Add it to the list
    }
    return slotList; // Return the SaveSlot list
}
```

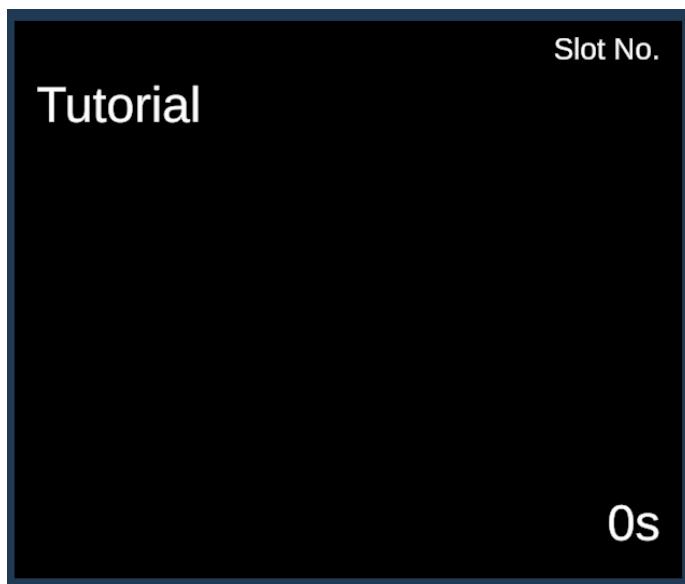
Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	The 'Saves' folder filled with slots	The player will have at least 1 save slot on a typical opening of the game	A SaveSlot list is returned with the correct information	Passed	
Boundary	The 'Saves' folder is empty	This can occur if the player opens the game, but then closes it before creating a save	An empty SaveSlot list is returned	Passed	
Error	The folder	This will occur	'Saves' is	Passed	

	'Saves' is not created	on opening the game for the first time, and as such must be handled properly	created		
--	------------------------	--	---------	--	--

Since this works properly, the code necessary to run the GUI needs to be implemented now. This code will use `ListAllSlots()` and then assign this data to the appropriate slots on a prefab, which would mimic the design made above. These elements are placed in a certain order, which allows them to be accessed using the `transform.GetChild()` method, provided you know the index that they are in the inspector.



This results in the below layout:



To access these, first the prefab must be instantiated into an object, and the reference to the prefab is stored so that its children components can have their values set in order. The whole process is repeated for each slot in the returned list, which are all added to the Scroll View UI object in Unity. This must also be run when the script is initialised so it will be ready when the player opens this menu upon opening the game.

```

using System.Collections.Generic;
using UnityEngine;
using TMPro;
using System;

public class GUISlots : MonoBehaviour
{
    [SerializeField] GameObject slotPrefab;

    private List<SaveSlot> slots;
    private void Start()
    {
        DisplaySaves();
    }
    public void DisplaySaves()
    {
        slots = SaveSystem.ListAllSlots();
        foreach (SaveSlot slot in slots)
        {
            GameObject slotInList = Instantiate(slotPrefab, gameObject.transform);
            slotInList.transform.GetChild(1).GetComponent<SelectSlot>().dataPath = slot.dataPath;
            TMP_Text number = slotInList.transform.GetChild(2).gameObject.GetComponent<TMP_Text>();
            TMP_Text time = slotInList.transform.GetChild(3).gameObject.GetComponent<TMP_Text>();
            TMP_Text scene = slotInList.transform.GetChild(4).gameObject.GetComponent<TMP_Text>();
            number.text = slot.number;
            time.text = TimeSpan.FromSeconds(slot.time).ToString();
            scene.text = slot.sceneName;
        }
    }
}

```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	The 'Saves' folder filled with slots	The player will have at least 1 save slot on a typical opening of the game, and this should be displayed properly	The right number of saves are seen in the scroll view	Passed	
Boundary	The 'Saves' folder is empty	This can occur if the player opens the game, but then closes it before creating a save	No saves are seen	Passed	

To allow the player to make a new save slot, there are a few things the system needs to do, similarly, to creating a save. The first check is to check if the top-level 'Saves' folder exists, and if it doesn't, it needs to be created. The internals of the folder are sorted to

check what the new slot number is, using the same code as before. Then, the potential directory of the slot needs to be checked to not exist, since if it already does, then something has gone wrong with the system. Provided it doesn't already exist, the folder is created and within it, the 'Auto' and 'Manual' folders are created as well. The first save file is then created and written within, at which point the lookup is edited by appending the necessary string to the file.

```

public static SaveData NewSave()
{
    if (!Directory.Exists(Path.Combine(Application.persistentDataPath, "Saves"))) // If Saves doesn't exist
    {
        Directory.CreateDirectory(Path.Combine(Application.persistentDataPath, "Saves")); // Create it
    }
    string[] dirs = Directory.GetDirectories(Path.Combine(Application.persistentDataPath, "Saves")); // Get all Save slots
    Array.Sort(dirs, new CompareSaves()); // Sort the slots using the same system
    int dirNum = 1;
    if (dirs.Length > 0)
    {
        string lastDir = dirs[1]; // Get the last directory
        dirNum = Int32.Parse(Regex.Match(lastDir, @"\d+$").ToString()) + 1; // Get the number of the last slot
    }
    // Create the folders for the save slot
    string folderName = "Slot" + dirNum.ToString(); // Create the folder name
    if (!Directory.Exists(Path.Combine(Application.persistentDataPath, "Saves", folderName))) // If the folder doesn't exist
    {
        Directory.CreateDirectory(Path.Combine(Application.persistentDataPath, "Saves", folderName)); // Create the folder
        Directory.CreateDirectory(Path.Combine(Application.persistentDataPath, "Saves", folderName, "Manual")); // Create the Manual folder inside
        Directory.CreateDirectory(Path.Combine(Application.persistentDataPath, "Saves", folderName, "Auto")); // Create the Auto folder inside
    }
    else
    {
        throw new Exception($"The save folder {dirNum} was unable to be created.");
    }
    // Create the save file
    SaveData data = new SaveData(); // Create new save data
    string path = Path.Combine(Application.persistentDataPath, "Saves", folderName, "Manual", "Slot" + dirNum.ToString() + "Manual1.dat"); // Get the path to the file
    using (StreamWriter saveWriter = new StreamWriter(path)) // Make the file on the system
    {
        string json = JsonConvert.SerializeObject(data, Formatting.Indented); // Convert the save data into JSON
        saveWriter.Write(json); // Fill with constructed save data
    }
    // Modify the lookup
    if (!File.Exists(Path.Combine(Application.persistentDataPath, "SaveLookup.txt"))) // If the lookup doesn't exist
    {
        File.Create(Path.Combine(Application.persistentDataPath, "SaveLookup.txt")).Close(); // Make it and then close it
    }
    File.AppendAllText(Path.Combine(Application.persistentDataPath, "SaveLookup.txt"), $"{folderName},{path}\n"); // Append the necessary pair
    return data; // Return the save data
}

```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Making a new save slot when the folder (Saves) has some already	The player will have at least 1 save slot on a typical opening of the game, and this should be displayed properly	The number of save slots increases by one, and is displayed in the scroll view	Partial: the entire list was appended on, leading to duplicated slots	Caused by not removing the previous slots in the menu
Boundary	Making a new save slot when the folder (Saves) is empty	This will be the state of the player's game on the first opening of the game	The new save slot is shown as the only one	Passed	

This display is caused by DisplaySaves not first deleting all the slots in the GUI before remaking them, and as such this tweak is needed. Importantly, the for loop needs to run in reverse to make sure that there are no NullReference errors.

```
public void DisplaySaves()
{
    for (int i = gameObject.transform.childCount - 1; i >= 0; i--)
    {
        Destroy(gameObject.transform.GetChild(i).gameObject);
    }
    slots = SaveSystem.ListAllSlots();
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Making a new save slot when the folder (Saves) has some already	The player will have at least 1 save slot on a typical opening of the game, and this should be displayed properly	The number of save slots increases by one, and is displayed in the scroll view	Passed	

Both saving and loading works, but the system itself isn't exactly finished. This is because the data needs to be actually applied to the game itself, where the data is spread around everywhere. This is done via the game's manager, which takes in the save data that had been selected and reads and applies each value to the right locations, such as the health to the player's Entity script. Once again, the main difficulty comes from the weaponry, since the ID of the object can't easily be used to get the prefab GameObject that has it.

This can be solved with the Addressables package that comes with Unity. It can be used to load in an object at any moment, provided you know the name of the object, and that object is marked as addressable. The package will handle the finding of the object in the game files, which is much better than having another script store references to all of the different weapons. Disadvantages of trying to do it with a script instead include:

- Every weapon must be loaded in if a GameObject reference is used, otherwise this will be marked as null and not work.
- If the path to the weapon prefab is stored instead, this will end up failing when the game is built since Unity changes the paths and name of assets when it compresses them into different locations and formats. This can be combated by

using the StreamingAssets folder instead, but this increases the size of the game overall since these objects are not compressed and hence is not preferable.

The main downside of using this combination of the Addressables package and the ID of the object is that the prefab name is required to be the same as the ID of the object. This may cause some issues for Henry Master's company, and as such I confirmed if this was ok before implementing this.

### *Interview*

**Stakeholder:** Don't worry about this downside, we were planning to do the same anyway as this makes managing the items far easier on our end.

Stakeholder Signature:



Since I have the go-ahead on implementing the system like so, the first step is to add the weapons to the package's watch, along with the folders that they are in as a backup.

Assets/Weapons/Guns	Assets/Weapons/Guns	Weapons
Assets/Weapons	Assets/Weapons	Weapons
Assets/Weapons/Blades	Assets/Weapons/Blades	Weapons
Assets/Weapons/Explosives	Assets/Weapons/Explosives	Weapons
Assets/Weapons/Bullets/Bullet.prefab	Assets/Weapons/Bullets/Bullet.prefab	Weapons
Assets/Weapons/Bullets	Assets/Weapons/Bullets	Weapons

With the system now having acknowledged the prefabs, I can go through each of the WeaponData in the save and use the ID to load in the object. Importantly, this will also be done with the asynchronous version of the method, to make sure that if the player has a more powerful computer, they will be able to take advantage of its capabilities. This also allows the intensive loading operation to take place in the background and let the rest of the game load in.

```

void ApplyData()
{
    // When a scene loads, it reloads the save data
    player.GetComponent<PlayerEntity>().Health = saveData.playerData.health; // Apply the player's health
    player.GetComponent<PlayerEntity>().MaxHealth = saveData.playerData.maxHealth; // Apply the player's maximum health
    player.transform.position = saveData.playerData.position; // Apply the player's position
    player.GetComponent<Rigidbody>().linearVelocity = saveData.playerData.velocity; // Apply the player's velocity
    player.transform.rotation = saveData.playerData.bodyRotation; // Apply the player's body's rotation
    Camera.main.transform.rotation = saveData.playerData.lookRotation; // Apply the player's vertical looking direction
    // Grab the prefab in the right order for the weapons
    foreach (WeaponData data in saveData.playerData.weaponSlots)
    {
        Addressables.LoadAssetAsync<GameObject>(data.name).Completed += (asyncOp) =>
        {
            // Load the weapon based on the ID
            if (asyncOp.Status == AsyncOperationStatus.Succeeded)
            {
                // If the load succeeded
                GameObject weapon = Instantiate(asyncOp.Result); // Create the weapon
                player.GetComponent<PlayerGunInteraction>().AddWeapon(weapon); // Add the weapon to the player
                weapon.GetComponent<Weapon>().TotalAmmo = data.totalAmmo; // Set the weapon's total ammo
                weapon.GetComponent<Weapon>().AmmoInClip = data.ammoInClip; // Set the remaining ammo in the weapon's clip
                weapon.GetComponent<Weapon>().ClipCapacity = data.clipCapacity; // Set the maximum capacity of the clip in the weapon
            }
            else
            {
                Debug.LogError("Failed to load weapon");
            }
        };
    }
    player.GetComponent<PlayerGunInteraction>().activeWeaponSlot = saveData.playerData.activeWeaponSlot; // Set the active weapon
    totalKills = saveData.playerData.totalKills; // Set the player's total kills
}

```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Normal SaveData	This is the standard action that will call	A SaveSlot list is returned with the correct information	Passed	
Error	The weapon ID is incorrect	This may occur when the game developer puts in the incorrect ID for the prefab in the WeaponType property in the Weapon script	An error is logged, and the weapon is not loaded in.	Passed	

## Menus and level transitions

The player will have to interact with many menus as they are playing the game. In Unity, UI elements are placed onto Canvas elements, and these can be disabled to hide the elements on that Canvas. To handle this behaviour, a separate dedicated manager can be created for this, which I shall call it ‘MenuManager’. It will store a reference to every single Canvas in the game in a dictionary, with its name being the key to access it. This

will allow for much faster access to the different Canvas objects, especially when SillyAustralian adds more in their own development.

The main problem this system has to solve is the fact that the references to the Canvas' may exist in other scenes, and as such the system has to be able to handle the inability to fetch the reference to every single Canvas. To start with, all the Canvas objects are attempted to be fetched using `GameObject.Find()`. However, the code below causes a `InvalidOperationException` error to occur.

```
! [12:41:45] InvalidOperationException: Collection was modified; enumeration operation may not execute.
System.Collections.Generic.Dictionary`2+KeyCollection+Enumerator[TKey,TValue].MoveNext () (at <ed969b0e627d471da4848289f9c322df>:0)
```

```
public void FetchCanvases(Scene __, LoadSceneMode __){
    foreach (string canvas in currentCanvas.Keys){
        currentCanvas[canvas] = GameObject.Find(canvas).GetComponent<Canvas>();
    }
}
```

Upon consulting the C# documentation, when the contents of a dictionary change, this operation will fail, since `Dictionary.Keys` does not return a copy of the keys but is in fact a reference to the actual set of keys. This error is therefore returned as a warning in case the user removed a key from the dictionary. To fix it, we need to force C# to return a copy rather than a reference by using `ToList()`, which does what we want due to the data structure casting.

```
public void FetchCanvases(Scene __, LoadSceneMode __){
    foreach (string canvas in currentCanvas.Keys.ToList()){
        currentCanvas[canvas] = GameObject.Find(canvas).GetComponent<Canvas>();
    }
    DisableAll();
}

public void DisableAll(){
    foreach (Canvas canvas in currentCanvas.Values.ToList()){
        if (canvas != null){
            canvas.gameObject.SetActive(false);
        }
    }
}
```

This error would also have shown up in `DisableAll()` which resets all canvases as a just-in-case for if any canvases were left enabled in between scene changes.

```
! [12:48:47] NullReferenceException: Object reference not set to an instance of an object
MenuManager.FetchCanvases (UnityEngine.SceneManagement.Scene __, UnityEngine.SceneManagement.LoadSceneMode __) (at Assets/Scripts/Managers/MenuManager.cs:45)

void Start()
{
    FetchCanvases(new Scene(), new LoadSceneMode()); // Fetch all the canvases
    SceneManager.sceneLoaded += FetchCanvases; // Attach the fetching of the canvases to every scene Load
}
```

This error seems to be caused by FetchCanvasi() being run in Start(). This might be because Start(), against what I previously thought, is running before the scene actually loads fully. However, upon moving this to Update(), the error still persisted. A Debug.Log() was added to see which Canvas was causing the error, and it turned out to be ‘Slots’, which had been set to be inactive. Consulting the documentation, I discovered that GameObject.Find() will not return an inactive object, and this was causing the error. In fact, there exists no built-in function within Unity that will return inactive objects, and as such I have to make my own function using the object list in the scene data:

```
public GameObject FindGameObject(string name){  
    List<GameObject> list = new List<GameObject>();  
    SceneManager.GetActiveScene().GetRootGameObjects(list);  
    foreach(GameObject go in list){  
        if (go.name == name){  
            return go;  
        }  
    }  
    return null;  
}
```

It functions similarly to GameObject.Find() but is more inefficient as it goes through the scene data. However, nonetheless it should help with the issue.

Indeed, it gets past ‘Slots’, which was previously causing the issue, but then gets stuck again at ‘Pause’. This is because ‘Pause’ is an object that has been marked as DontDestroyOnLoad, meaning that it does not exist in any of the following scene’s data. The typical way to grab references to objects that are marked as DontDestroyOnLoad is to have passed a manual reference in the Unity Inspector, but this is not feasible in this scenario.

Instead, I came up with a better solution, using the CanvasGroup component. This allows you to hide and show objects without disabling it. It also allows you to control whether a canvas is interactable or not, which means the Dictionary would have to be modified to use a tuple as well to store whether a menu should be interactable or not (this is needed because of the player’s HUD).

```
currentCanvas = new Dictionary<string, (Canvas, bool)>(){
    {"Main", (main, true)},
    {"Slots", (slots, true)},
    {"Pause", (pause, true)},
    {"HUD", (hud, false)},
    {"Options", (options, true)},
    {"Death", (death, true)}
```

This slightly modified dictionary now stores the Canvas as part of a tuple with the intractability state of the Canvas stored as well. This is necessary for on-screen elements, such as the HUD, that the player should not be able to click. This requires slight refactoring of the code to allow for changes in the dictionary.

The way other system interact with the menu manager is by calling either ChangeMenu() or GoBack(). ChangeMenu() requires the name of the menu that needs to be switched to, and importantly, it disables the ability for the Canvas to block the mouse movement inputs and the mouse click inputs, along with setting the alpha for that Canvas to 0, making it invisible. lastMenu is now updated to store the value of currentMenu. Next, the new menu is set to be visible by setting its alpha to 1. If the Canvas has been marked as interactable, then it is allowed to detect and hence block the mouse's inputs and movements from passing through to anything else in the game. Finally, it sets the current menu to the one that had been passed as an argument.

```
public void ChangeMenu(string menu)
{
    if (currentMenu != null)
    {
        // Disable the current menu if it is exists
        currentCanvas[currentMenu].Item1.GetComponent<CanvasGroup>().interactable = false; // Make the player unable to interact with the menu
        currentCanvas[currentMenu].Item1.GetComponent<CanvasGroup>().blocksRaycasts = false; // Make it invisible to all inputs
        currentCanvas[currentMenu].Item1.GetComponent<CanvasGroup>().alpha = 0; // Make it invisible to the player
        lastMenu = currentMenu; // Sets the previous menu to the current one
    } // Enable the new canvas
    if (currentCanvas[menu].Item2)
    {
        currentCanvas[menu].Item1.GetComponent<CanvasGroup>().interactable = true; // Make it interactable if it is supposed to be
        currentCanvas[menu].Item1.GetComponent<CanvasGroup>().blocksRaycasts = true; // Make it possible for inputs to sense it if you should be able to interact with it
    }
    currentCanvas[menu].Item1.GetComponent<CanvasGroup>().alpha = 1; // Make it visible to the player
    currentMenu = menu; // Changes the current menu
}

public void GoBack()
{
    if (lastMenu != null)
    {
        // If there is a previous menu and it exists in the scene.
        currentCanvas[currentMenu].Item1.GetComponent<CanvasGroup>().interactable = false; // Make the player unable to interact with the menu
        currentCanvas[currentMenu].Item1.GetComponent<CanvasGroup>().blocksRaycasts = false; // Make it invisible to all inputs
        currentCanvas[currentMenu].Item1.GetComponent<CanvasGroup>().alpha = 0; // Make it invisible to the player
        if (currentCanvas[lastMenu].Item2)
        {
            currentCanvas[lastMenu].Item1.GetComponent<CanvasGroup>().interactable = true; // Make it interactable if it is supposed to be
            currentCanvas[lastMenu].Item1.GetComponent<CanvasGroup>().blocksRaycasts = true; // Make it possible for inputs to sense it if you should be able to interact with it
        }
        currentCanvas[lastMenu].Item1.GetComponent<CanvasGroup>().alpha = 1; // Make it visible to the player
        string temp = currentMenu; // store the old menu
        currentMenu = lastMenu; // Swap the current menu with the last menu
        lastMenu = temp; // set the previous menu to the old one
    }
    else
    {
        Debug.LogWarning("No back menu option set!");
    }
}
```

The above code has a minor issue, in that GoBack() shares a lot of duplicate code with ChangeMenu(). The only major changes are that menu is replaced with lastMenu, and

that lastMenu and currentMennu are swapped at the end. This leads to an optimisation where lastMenu is passed as the argument for ChangeMenu(). ChangeMenu() will then disable the currently activated Canvas, and then enable the Canvas named by lastMenu. ChangeMenu() has swapped the values of lastMenu and currentMenu, and the only reason this works is that the parameter acts as a copy for the last menu. Since there are 3 variables, of which are arranged in the right order, it is possible to swap the 2 relevant variables.

```
public void GoBack()
{
    if (lastMenu != null)
    { // If there is a previous menu and it exists in the scene.
        ChangeMenu(lastMenu); // change to the previous menu
    }
    else
    {
        Debug.LogWarning("No back menu option set!");
    }
}
```



[16:52:59] MissingReferenceException: The object of type 'UnityEngine.Canvas' has been destroyed but you are still trying to access it.  
Your script should either check if it is null or you should not destroy the object.

However, ChangeMenu() and GoBack() both have a problem, which leads to the error above. This occurs because, by default, the string object cannot be set to null. But a string can be set to be empty, and I can scan for this instead. Then to double down on the problem, I can add a check in to see whether the Canvas in that dictionary now equals null, since when the object unloads any references to it are replaced with null.

```
public void ChangeMenu(string menu){
    if (currentMenu != "" && currentCanvas[currentMenu].Item1 != null){
```

Another optimisation can be made to minimise the amount of if statements required. Since only the Canvases that were supposed to be interactable are set to be interactable, I can set this state of the canvas to be equivalent to the value in the dictionary.

```
currentCanvas[menu].Item1.GetComponent<CanvasGroup>().interactable = currentCanvas[menu].Item2; /
currentCanvas[menu].Item1.GetComponent<CanvasGroup>().blocksRaycasts = currentCanvas[menu].Item2;
```

To test both of these functions, they will be activated by buttons on the screen that will call the relevant menu that they want to be activated.

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Selecting a new menu	This is the standard operation this	The current menu will be disabled and	Passed	

		system will experience and hence is suitable for a valid test	not interactable		
Valid	Going back	This will make sure that the lastMenu variable is working appropriately	The previous menu will be reenabled and the current one will be disable	Passed	
Erroneous	Requesting a menu that isn't in the dictionary	In case the developer puts in a wrong system, the system should not crash	A debug message is logged with the error and the current menu stays up	Failed: no code was written to handle this	

To handle the error, we need to validate the input handed into the function against all keys in the dictionary.

```
if (!currentCanvas.Keys.Contains(menu)){ // If the menu is not in the dictionary keys
    Debug.LogError($"The menu '{menu}' is not in the registered list of accepted menus");
}
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Erroneous	Requesting a menu that isn't in the dictionary	In case the developer puts in a wrong menu, the system should not crash	A debug message is logged with the error and the current menu stays up	Passed	

The death screen will be activated when the player dies and hence will have slightly different requirements surrounding it. On the screen, it will allow the player to load their most recent save, which has already been implemented. However, to stop the player moving and continuing with their game, the player's controls will be disconnected from the movement controller. The player's Entity script will recognise when the player has died, and instead of destroying the player's game object, it will disable the Player input map, which contains all the controls for playing the game normally.

```
public override void Kill()
{
    PlayerManager.Instance.inputs.Player.Disable();
    PlayerManager.Instance.PlayerDeath();
    MenuManager.Instance.ChangeMenu("Death");
}
```

It will then tell the game manager that the player has died, which changes the player's movement state to be dead, which will set the player's velocity to 0, effectively freezing its movement.

```
public void PlayerDeath()
{
    player.GetComponent<PlayerMovement>().movementState = PlayerMovement.PlayerMovementState.dead; // Stop the player's movement
}
```

Finally, the current menu is changed to the death screen. When the player clicks the button to reload their game, it will run through the normal process, changing the current movement state to idle at the end of it to allow the player to move again.

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	The player dying	The player is expected to die when playing the game, and hence must be tested correctly	The player's movement stops, and they can no longer control the character, and the death screen is shown	Passed	
Valid	Clicking to reload the game	The player will need to reload their game after they die	The previous save is loaded just like a level load, and the player regains control	Passed	

On the main menu and on the pause menu, the player will also have the ability to quit the game. Quitting the game depends on whether the game is built or still within the editor. When within the editor, the play state of the editor needs to be set to false, while when the game is an executable instead, Application.Quit() needs to be called instead,

which relates to the process-kill system call. To avoid any potential errors with variables or functions not existing in either state, a preprocessor directive can be added. These are symbols which are used by C# to mark or do certain things while the file is being compiled. In this scenario, if the compiler is being run within the Unity Editor, the play state being set to false will be the only line of code inside Quit(); otherwise, the system call will be the only code inside Quit().

```
public void Quit()
{
#if UNITY_EDITOR // If this function is being run inside the Unity Editor
    UnityEditor.EditorApplication.isPlaying = false; // Set the current playing state to false
#else
    Application.Quit(); // Call the system-level quit on the game
#endif
}
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	The button is pressed when the game is in the editor	The player will eventually want to close the game, and as such this ease-of-use method will be required.	The editor's play state is set to false, and the preview stops	Passed	
Valid	The button is pressed when the game is an executable	The player will eventually want to close the game, and as such this ease-of-use method will be required.	The game will close and leave the desktop	Passed	

The player should also be able to see their health on the HUD, which consists of only a slider. At maximum health, the slider will be full, while at any other value, the slider will be as full as the proportion of health remaining. For ease-of-use, there will also be a number on top dictating the amount of health remaining.

```
private void Update()
{
    slider.value = Health / MaxHealth;
    healthText.text = Health.ToString();
}
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Running into	This is a basic	The player	Passed	

	the spike	test for damage as well as well as verifying environmental damage and whether the player's health is updated on the HUD	will take damage as is detailed, and this will be shown on the HUD		
Boundary	Running into a spike with 1 HP (or enough for the next hit to kill the player)	This will test whether the kill function works	The death screen will show up	Passed	

This system works fine for most of the UI elements in the game, but there is one exception to this that needs to be handled by itself – the level loading screen. This is only enabled when a new scene needs to load in and takes priority over every other screen. The difficulty that needs to be handled with it is that displaying a loading bar should not stop the level itself from loading, and therefore one of them has to run asynchronously. Unity does have a built-in variation of its LoadScene() function that loads a scene asynchronously, that being LoadSceneAsync(). However, just calling this function with the name of the level and then updating the progression value will not allow the change to be visible on the player's screen. If I did do this, then the surrounding function will wait for it to complete. Instead, I need to wrap it in a function that will also be asynchronous, and in the version of C# that Unity uses, this can be done by defining a function's return type as an IEnumerator. This way, it is marked to hand back control flow to the rest of the program and then come back to where it was afterwards. Doing it like this is better than instead running this code inside of the Update() function as the progress of the scene being loaded may change as the other scripts are being run, and as such the progress of the scene being loaded will not be accurate when run through Update().

```

using System.Collections;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

public class LevelLoading : MonoBehaviour
{
    Slider progressBar;
    private void Start()
    {
        progressBar = transform.GetChild(0).GetComponent<Slider>();
    }

    public void Load(string sceneName, GameObject caller)
    {
        StartCoroutine(DisplayProgress(sceneName));
    }

    IEnumerator DisplayProgress(string sceneName)
    {
        AsyncOperation load = SceneManager.LoadSceneAsync(sceneName);
        while (!load.isDone)
        {
            progressBar.value = load.priority;
            yield return null;
        }
    }
}

```

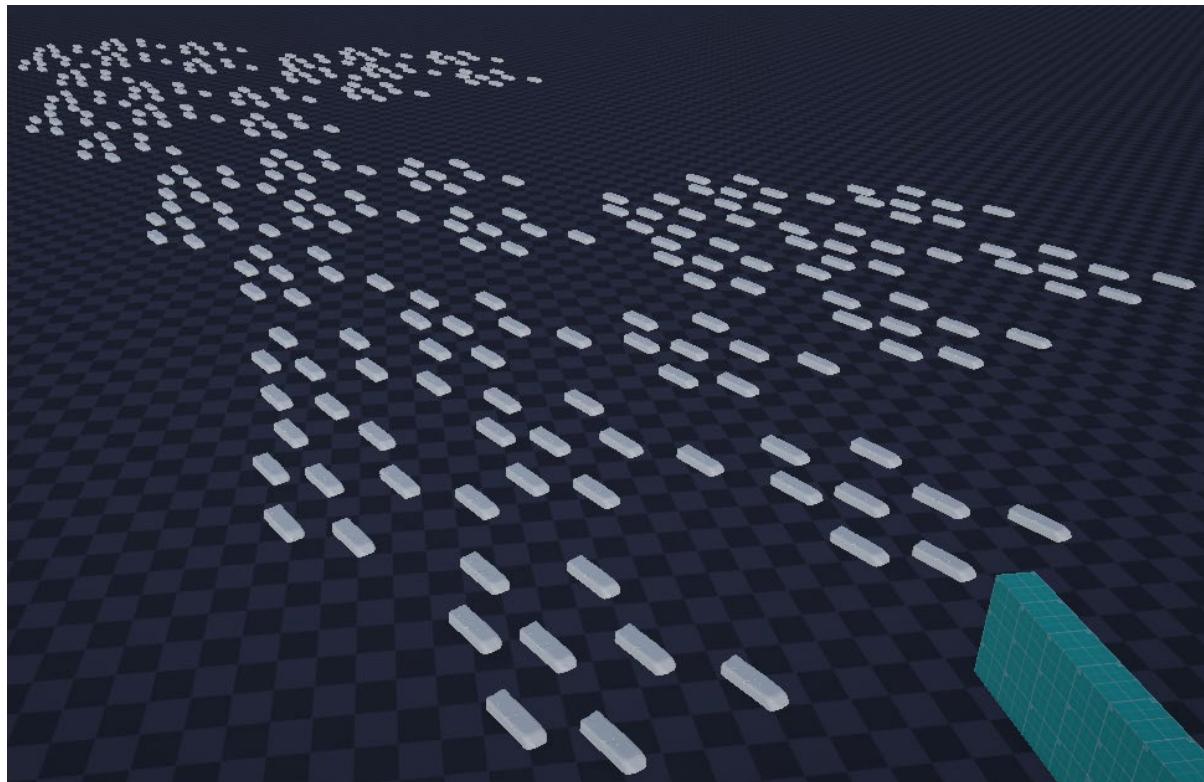
To run this function, the button links to a separate function in the game manager. Within this, it first checks whether the scene exists. If it does, all the Canvas objects are disabled, following which, the loading screen is enabled. Once enabled, the scene is requested to load, passing program flow to the coroutine. If the scene did not exist, an error is logged instead for a developer to view.

```

public void LoadSlot(string sceneName)
{
    if(Application.CanStreamedLevelBeLoaded(sceneName)){ // Check if the level exists
        menuManager.DisableAll();
        sceneLoader.gameObject.SetActive(true); // Enables the load screen
        sceneLoader.Load(sceneName); // Once the new scene has loaded, execution will return here
    }
    else
    {
        Debug.LogError($"Scene '{sceneName}' does not exist.");
    }
}

```

Testing this function is quite difficult as I do not have any scenes that are difficult enough to load that they need more than a frames worth of time. As such, I sourced one of my own models that was extremely dense in the number of polygons it had. Then, by duplicating this many times, the scene would gain many polygons, which would slow down the loading of the scene. This would emulate a larger, much fuller scene, and with this I was able to test it correctly.

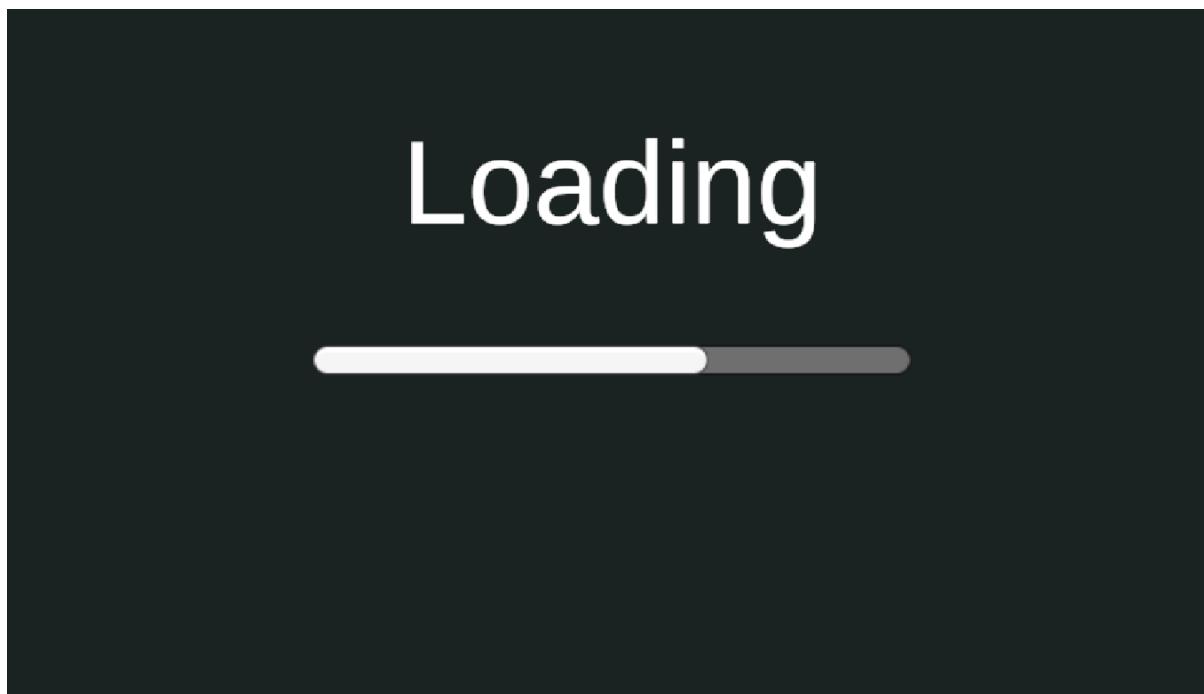


Batches: 1059 Saved by batch  
Tris: 766.7M Verts: 389.6M  
Screen: 922x519 - 5.5 MB

*(The number of rubbers placed as an attempt to slow down the scene, as well as the polygon count)*

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Selecting to load a save	This will be the easiest way to test the loading screen and makes a suitable test since this tests the	The current menu will be disabled, and the loading screen will come up	Partial: the progress bar did not reach one, stopping at 90% instead	

		function that runs the screen			
Erroneous	Loading a scene that doesn't exist	In case the save file is wrong, the system should not crash	A debug message is logged with the error and the current menu stays up	Passed	



According to the documentation, the progress of the load will never reach 1, as it is split into 2 sections: the first 90% is loading the assets of the level, while the last 10% is the unloading of the current level and replacing it with the new one. Unloading the current level includes the loading screen itself, so we need to change the progress value shown on the slider to being 1 when `load.progress` is at 0.9. This can be done by simply diving the progress by 0.9, resulting in the necessary scaling.

```
float progress = Mathf.Clamp01(load.progress / 0.9f);
progressBar.value = progress;
```

`Mathf.Clamp01()` makes sure that the value provided is between 0 and 1, and this makes sure that just in case the value for `load.progress` goes above 0.9, the value does not break the slider.

Type	What data fits criteria	Justification	Expected	Actual	Comments

Valid	Selecting to load a save	This will be the easiest way to test the loading screen and makes a suitable test since this tests the function that runs the screen	The current menu will be disabled, and the loading screen will come up	Partial: the progress bar did not reach one, stopping at 90% instead	
-------	--------------------------	--	--	--	--

Another thing that needs to be taken into account with switching scenes is the state of each scene. Within certain scenes, the player may not be allowed to move in straight away and this needs to be set before the player gains control. Scenes will also have a start position and a UI screen that needs to be open first, and all these pieces of information will be kept within a database. Each of these properties are stored in separate fields, where the primary key is the scene name.

To start with, I had to learn how to use SQL in C#. The database did not need to be hosted online, and as such SQLite was a perfectly usable solution of the SQL dialects. It was also an advantage to use this since I had some experience with using this in Python before and was a somewhat well-used solution in Unity projects. However, the drawback is that SQLite was overall the least-used SQL dialect, as many flock to using MySQL instead. However, the many variations in syntax and its main use as a multi-user access database system is not a capability I require. In fact, SQLite is extremely useful in being lightweight, and its main use is for small applications such as this game.

Having decided on a dialect to use, I searched around for how this was implemented within Unity or C#. Unity contains a plugin within it called Mono.Data.Sqlite and combined with System.Data from C#, I could create database connections using the IDbConnection interface and send commands using the IDbCommand interface.

```

public bool GetMoveability(string sceneName){
    db = OpenDb();
    IDbCommand request = db.CreateCommand(); // Begins a query
    request.CommandText = $"SELECT moveability FROM SceneInfo WHERE name='{sceneName}'"; // Get whether the player's movement script should be enabled or not
    IDataReader response = request.ExecuteReader(); // Execute the query
    bool success = response.Read(); // Read if the query succeeded
    if (!success){
        Debug.LogWarning($"Scene '{sceneName}' does not have an entry in the info database for the player's moveability.");
        return true; // Most of the time the scene will be a Level, so it is safer to put true.
    }
    db.Close();
    return response.GetInt32(0) != 0; // return whether the first integer result is equal to 0
}
public string GetCanvasOpen(string sceneName){
    db = OpenDb();
    IDbCommand request = db.CreateCommand(); // Begins a query
    request.CommandText = $"SELECT menuOpen FROM SceneInfo WHERE name='{sceneName}'"; // Get what menu should be open
    IDataReader response = request.ExecuteReader(); // Execute the query
    bool success = response.Read(); // Read if the query succeeded
    if (!success){
        Debug.LogWarning($"Scene '{sceneName}' does not have an entry in the info database for the canvas to be open.");
        return "HUD"; // Most of the time the scene will be a Level, so it is safer to return HUD, especially since this is the default value of the database.
    }
    db.Close();
    return response.GetString(0); // Return the first string result
}
private Vector3 GetStartPos(string sceneName){
    db = OpenDb();
    IDbCommand request = db.CreateCommand(); // Begins a query
    request.CommandText = $"SELECT startX, startY, startZ FROM SceneInfo WHERE name='{sceneName}'"; // Get where the player should spawn in from
    IDataReader response = request.ExecuteReader(); // Execute the query
    bool success = response.Read(); // Read if the query succeeded
    if (!success){
        Debug.LogWarning($"Scene '{sceneName}' does not have an entry in the info database for the start position.");
        return Vector3.zero; // Default backup as most Levels should be centered near 0,0,0
    }
    db.Close();
    return new Vector3(response.GetFloat(0), response.GetFloat(1), response.GetFloat(2)); // Return the combination of the 3 axis
}

```

This first run through can be optimised to have a general query function that takes in the scene name and the field that the function needs to select, which also handles connecting to the database. Then the other functions are replaced as wrapper functions that feed this querying function and then read the right type of data from the response or report the correct error message.

```

private (IDataReader, bool) QueryDatabase(string sceneName, string field){
    db = OpenDb(); // Open and store a reference to the database
    IDbCommand request = db.CreateCommand(); // Begins a query
    request.CommandText = $"SELECT {field} FROM SceneInfo WHERE name='{sceneName}'"; // Get whether the player's movement script should be enabled or not
    IDataReader response = request.ExecuteReader(); // Execute the query
    bool success = response.Read(); // Read if the query succeeded
    if (!success){
        Debug.LogWarning($"Scene '{sceneName}' does not have an entry in the info database for the player's {field}.");
    }
    return (response, success); // Most of the time the scene will be a Level, so it is safer to put true.
}
public bool GetMoveability(string sceneName){
    (IDataReader response, bool success) = QueryDatabase(sceneName, "moveability"); // Query for the moveability
    db.Close(); // Close the database
    return success ? response.GetInt32(0) != 0 : true; // return whether the first integer result is equal to 0 if succeeded, otherwise return true since it will usually be a Level
}
public string GetCanvasOpen(string sceneName){
    (IDataReader response, bool success) = QueryDatabase(sceneName, "menuOpen"); // Query for the open canvas
    db.Close(); // Close the database
    return success ? response.GetString(0) : "HUD"; // Return the first string result if succeeded, otherwise return "HUD" since it will usually be a Level
}
private Vector3 GetStartPos(string sceneName){
    (IDataReader response, bool success) = QueryDatabase(sceneName, "startX, startY, startZ"); // Query for the start position
    db.Close(); // Close the database
    return success ? new Vector3(response.GetFloat(0), response.GetFloat(1), response.GetFloat(2)) : new Vector3(0,0,0.85f,0); // Return the combination of the 3 axis if succeeded or the database default otherwise
}

```

With this change, the function for setting the cursor state can be modified to also check the moveability of the player like below:

```

public void SetCursorLock(Scene scene, LoadSceneMode _)
{
    if (manager.GetMoveability(scene.name) && !MenuManager.Instance.GetCurrentCanvasInteractability())
    { // If the player can move, and the current canvas isn't interactable
        Cursor.lockState = CursorLockMode.Locked; // Lock the player's cursor
        Debug.Log("Locked cursor.");
    }
    else
    {
        Cursor.lockState = CursorLockMode.None; // Unlock the player's cursor
        Debug.Log("Unlocked cursor.");
    }
}

```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Loading a new scene which the player can't move in	This will be required most surely when the Main Menu is loaded in and will be the one thing every player will interact with.	The player will be unable to move, the cursor will be unlocked, the player will be in the right location, and the right menu is displayed	Passed	
Valid	Loading a scene which the player can move in	The player will need to load into a new level as they play the game, or when they load a save, and hence this must be tested properly	The player will be able to move, the cursor will be locked, the player will be in the right location, and the right menu is displayed	Passed	

## Miscellaneous

### *Referencing the Game Manager*

This is a minor change that was used to further optimise the code. Originally, the game manager was an object that could easily be referenced beforehand in each scene just by dragging the reference in the inspector. However, once multiple scenes were added, the manager was changed to an object that was only created once, then persisted

between the scene changes. Because of this, the inspector reference no longer worked, since the object would not exist in most scenes before the scene was loaded. As such, `GameObject.Find()` was first used as such to find the Game Manager object.

```
manager = GameObject.Find("Game Manager").
GetComponent<PlayerManager>();
```

`GameObject.Find()` is useful, however not every single object required to use it, since some other objects would also be taken along in the scene change, such as the player or certain UI elements. Therefore, you must first check if the manager reference is null, and if it is, fetch the reference.

```
if (manager == null){
    if (transform.parent.gameObject.name == "Game Manager"){
        manager = transform.parent.gameObject.
        GetComponent<PlayerManager>();
    } else {
        manager = GameObject.Find("Game Manager").
        GetComponent<PlayerManager>();
    }
}
```

`GameObject.Find()` is a very simple solution to the problem, but it comes with a few drawbacks, of which the main ones are that it is extremely slow (it uses a linear search to find the object) and it uses a string to reference the name of the object which is susceptible to typos. Therefore, the even better solution to reducing the time spent searching for the Game Manager is to not search for it at all and make it a static reference. This works since only 1 Game Manager exists in memory at any time and therefore no problems will occur. This method is called a Singleton and is commonly used for manager objects.

```
public class PlayerManager : MonoBehaviour
{
    public static PlayerManager Instance;
```

First the public reference is defined within the script, and then any script referencing the manager needs only to reference it like below.

```
private PlayerManager manager = PlayerManager.Instance;
```

This is the best solution since everything is baked into the code beforehand, which is the most efficient solution.

### *Referencing the camera*

While reading the Unity documentation, I discovered that Unity has a built-in static reference to the main camera. Since this project only uses 1 camera (the one for the player) it was easier, faster, and more efficient to use Camera.main instead of finding a reference to it, either by assigning it in the inspector or using GameObject.Find().

## End of prototype tests

### *Saving and Loading*

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Saving a new save with others already being made	The most typical action the player will be taking is this, and therefore it should be tested properly	A new save will be created with the newest number	Passed	
Valid	Loading a save file	It goes hand in hand with saving, and will also be a core function of the system	The save file is opened and the data is returned	Passed	
Boundary	Saving a file when none of the directories or files exist	This is necessary to test for since this will be the state of the user's system when they first launch the game	All the appropriate files and folders are created	Passed	
Error	Passing a file to load that doesn't exist	In case the lookup ends up corrupted, or the user	Error code is logged	Passed	

		deletes certain files, the function should not crash, and instead log the error			
Error	'SaveLookup.txt' is not created	This will occur when the player opens the game for the first time, and must be handled properly	The file is created	Passed	

*Save Data Application*

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Normal SaveData	This is the standard action that will call	A SaveSlot list is returned with the correct information	Passed	
Error	The weapon ID is incorrect	This may occur when the game developer puts in the incorrect ID for the prefab in the WeaponType property in the Weapon script	An error is logged, and the weapon is not loaded in.	Passed	

*Displaying Saves Normally*

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	The 'Saves' folder filled with slots	The player will have at least 1 save slot on a typical	The right number of saves are seen in the	Passed	

		opening of the game, and this should be displayed properly	scroll view		
Boundary	The 'Saves' folder is empty	This can occur if the player opens the game, but then closes it before creating a save	No saves are seen	Passed	

*Displaying Saves when making a new save slot*

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Making a new save slot when the folder (Saves) has some already	The player will have at least 1 save slot on a typical opening of the game, and this should be displayed properly	The number of save slots increases by one, and is displayed in the scroll view	Passed	
Boundary	Making a new save slot when the folder (Saves) is empty	This will be the state of the player's game on the first opening of the game	The new save slot is shown as the only one	Passed	

*Moveability and Cursor state when loading a scene*

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Loading a new scene which the player can't move in	This will be required most surely when the Main Menu is loaded in and will be the one thing every player will	The player will be unable to move, the cursor will be unlocked, the player will be in the right	Passed	

		interact with.	location, and the right menu is displayed		
Valid	Loading a scene which the player can move in	The player will need to load into a new level as they play the game, or when they load a save, and hence this must be tested properly	The player will be able to move, the cursor will be locked, the player will be in the right location, and the right menu is displayed	Passed	

## Interview

**Stakeholder:** Most of these systems are perfect and do not need any more work. However, I would like the loading screen to have a button, and until this button is clicked, it will not go to the next level. Also, an extra feature that I would like implemented is an options menu, which allows the player to change the volume of the game, as well as the camera's FOV

Stakeholder Signature:



## Prototype 4

### Button to loading screen

To implement this change, a particular flag on the loading function needs to be set, specifically `allowSceneActivation`, which disables the action of the scenes being swapped. This changes the necessary test in the while loop, from checking if `load.isDone == true` to `if progressBar.value < 1`. This is necessary since `isDone` never

gets set to true allowSceneActivation is set to false, but this doesn't mean that the check can't be load.progress < 0.9. The reason for using the progress bar's value instead is that due to the function being an IEnumerator, it may reach the done state before the progress bar is updated fully, and therefore the player may not see a full progress bar by the time the function exits. It is also necessary to change the yield statement to wait until the end of the frame to prevent a well-known crash caused by IEnumerators.

```
public void Load(string sceneName)
{
    button.gameObject.SetActive(false); // Make sure that the button is disabled
    StartCoroutine(DisplayProgress(sceneName)); // Start loading
}

public void ShowButton(){
    if (button == null){ // If the button is not assigned
        button = transform.GetChild(2).GetComponent<Button>(); // Assign the button
    }
    button.gameObject.SetActive(true); // Enable the button
}
public void FinishLoad() {
    op.allowSceneActivation = true; // Switch to the new scene
}

IEnumerator DisplayProgress(string sceneName)
{
    AsyncOperation load = SceneManager.LoadSceneAsync(sceneName); // Begin loading the scene in the background
    op = load; // Store so that the button can use later
    load.allowSceneActivation = false; // Make sure the level does not automatically replace the one
    Debug.Log($"Started scene load of scene '{sceneName}'");
    while (progressBar.value < 1) // While the bar has not finished - makes sure that the player sees a filled in bar
    {
        float progress = Mathf.Clamp01(load.progress / 0.9f); // Get the current progress

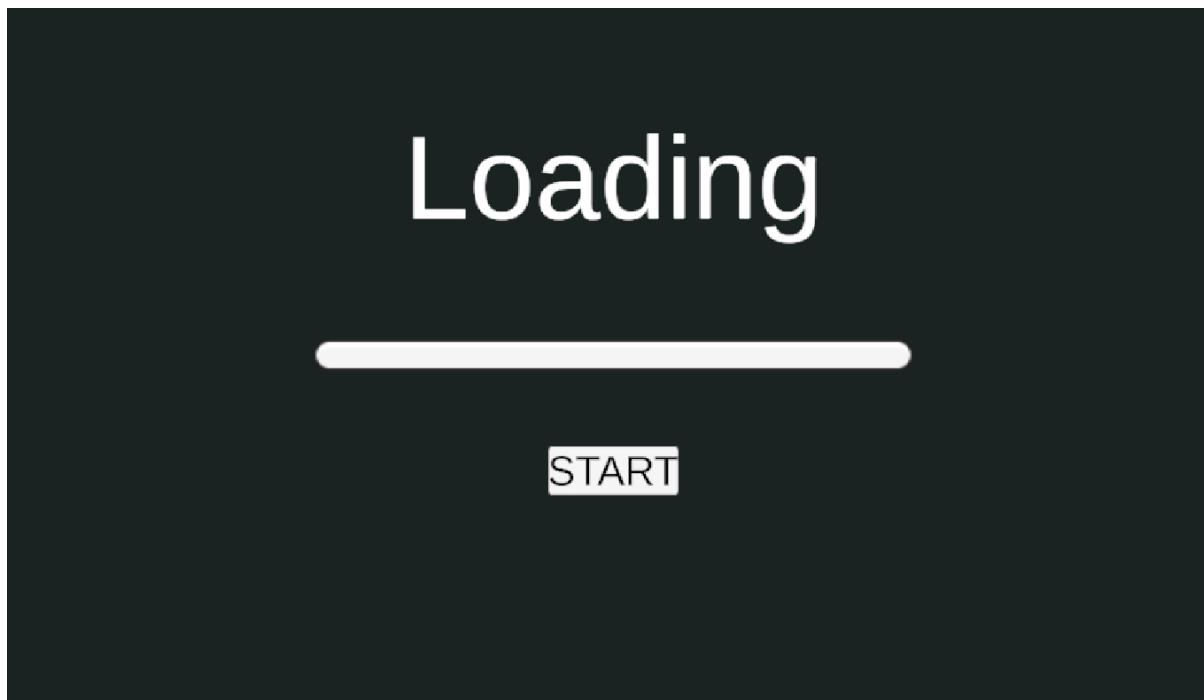
        progressBar.value = progress; // Assign the progress to the bar
        yield return new WaitForEndOfFrame(); // Wait for the end of frame
    }
    ShowButton(); // Show the button
}
```

When running this, after the first iteration, the script would lose the reference to the slider object. After much debugging, I was unable to find the reason for this, but it seemed to happen after the yield statement. As such, I placed the below check to fetch the reference if it were to be lost, and this solved the NullReferenceException that would occur from trying to read a value from the slider when the script couldn't find the object to read from.

```
if (progressBar == null){ // If the progress bar has been dereferenced
    progressBar = transform.GetChild(1).GetComponent<Slider>(); // Get the slider object
}
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Selecting to load a save	This will be the easiest way to test the loading screen and	The current menu will be disabled, and the loading screen will	Passed	

		makes a suitable test since this tests the function that runs the screen	come up. When the loading is completed, it will show the button with a full progress bar		
Erroneous	Loading a scene that doesn't exist	In case the save file is wrong, the system should not crash	A debug message is logged with the error and the current menu stays up	Passed	



(The screen when the level has finished loading)

#### Interview

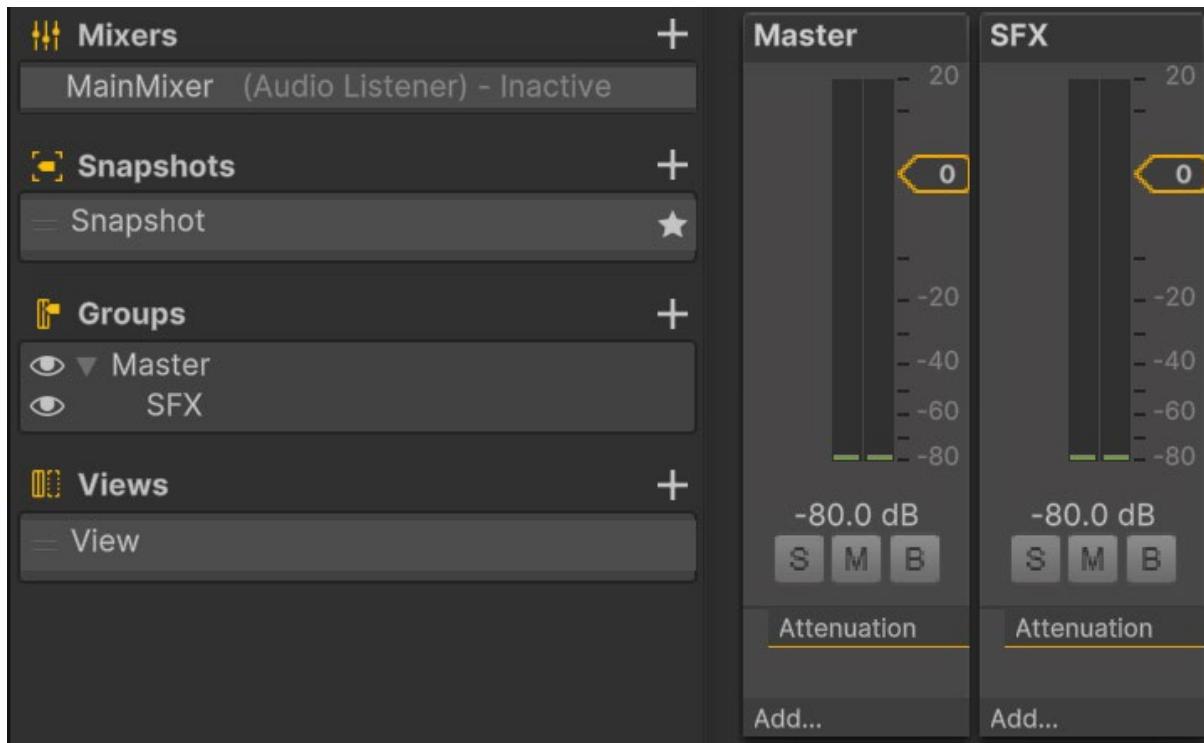
Stakeholder: This is better since it allows players time to rest between levels and not have to be at the computer constantly if they need to go somewhere else temporarily due to the worry of the level starting without them. Feel free to move on now.

Stakeholder Signature:

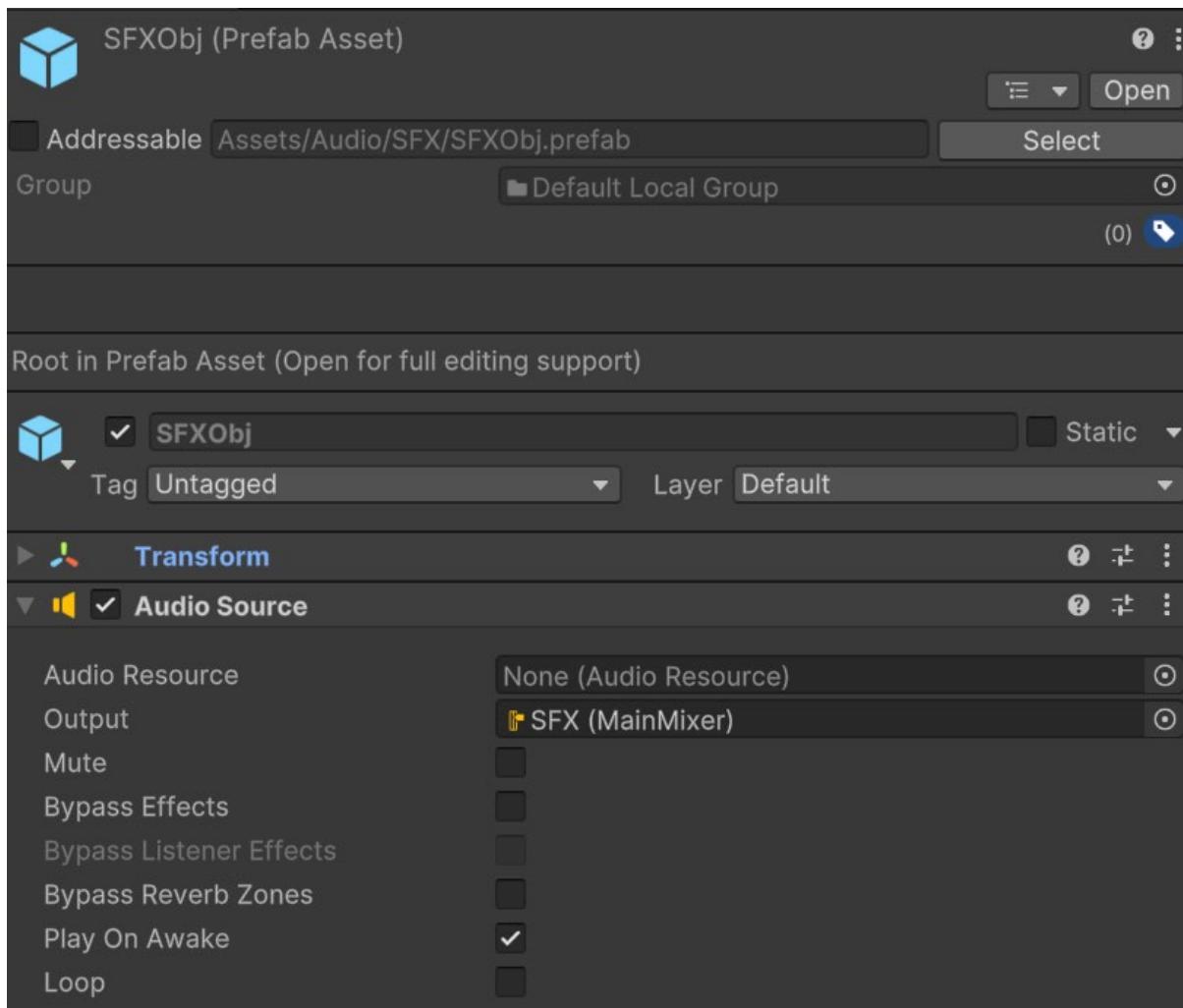


## Audio

Handling audio in the game is not that difficult, consisting mainly of using the built-in functions to play particular audio clips through a mixer object, which requires some setup within Unity itself. Importantly, for organisation, it is best to split audio into the different channels, and hence types, that they may run through, such as sound effects going through the SFX channel. The common channel that this gets routed through in the end is the Master channel which is applied last on top of every channel together and is typically used to control the volume of the game.



In Unity, audio files are played using an object with the ‘Audio Source’ component attached to it. By making this object into a prefab, it makes the process of playing these sound effects much easier than before. The Audio Source component must have the `playOnAwake` property set to true for this system to work.



All requests to play sound effects are then passed through the audio manager, which then instantiates the prefab at the location specified by the script, and assigns the clip to the object, allowing it to play. This object is then destroyed once the clip is finished to reduce the load on the system.

```
public void PlaySFXClip(AudioClip clip, Transform location, float volume)
{
    AudioSource source = Instantiate(sfxPrefab, location.position, Quaternion.identity); // Make a new prefab at the location specified with no rotation
    source.clip = clip; // Set the clip to start playing
    source.volume = volume; // Set the volume of the clip
    source.Play(); // Allow the clip to start playing
    Destroy(source.gameObject, source.clip.length); // Destroy the object once the clip ends
}
```

To test this system, I needed both a script to request the playing of sounds, along with sound effects to go alongside them.

To make these sound effects, I used a program I found called ChipTone. There were many other options, but this was the most user-friendly one that I found, that also allowed the easy generation of these sounds as required.



The 2 sounds that I created was one for hovering over a UI button and one for selecting/clicking the UI button. These are added to the Addressables package's watch and then can be loaded when the script begins.

```
void Awake()
{
    Addressables.LoadAssetAsync<AudioClip>("Assets/Audio/SFX/UIHover.wav").Completed += (asyncOp) =>
    { // Attempt to load the hover effect
        if (asyncOp.Status == AsyncOperationStatus.Succeeded)
        { // If the load succeeds
            uiHover = asyncOp.Result; // Assign the hover sound
        }
        else
        {
            Debug.LogError("Failed to load UI audio.");
        }
    };
    Addressables.LoadAssetAsync<AudioClip>("Assets/Audio/SFX/UISelect.wav").Completed += (asyncOp) =>
    { // Attempt to load the select effect
        if (asyncOp.Status == AsyncOperationStatus.Succeeded)
        { // If the load succeeds
            uiSelect = asyncOp.Result; // Assign the select sound
        }
        else
        {
            Debug.LogError("Failed to load UI audio.");
        }
    };
}
```

To run the hover sound effect, this script needs to play the SFX when the button is hovered over, which is automatically run by OnPointerEnter.

```
public void OnPointerEnter(PointerEventData eventData)
{
    // When the mouse hovers over the button
    AudioManager.Instance.PlaySFXClip(uiHover, transform, 1); // Play uiHover
}
```

Conversely, handling the click sound requires first listening to the onClick event fired by the button when it is pressed.

```
void Start()
{
    // Run OnClick when the button is clicked
    gameObject.GetComponent<Button>().onClick.AddListener(OnClick);
}

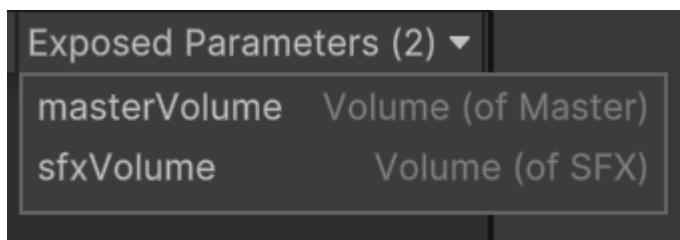
void OnClick()
{
    AudioManager.Instance.PlaySFXClip(uiSelect, transform, 1); // Play uiSelect
}
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Hovering over the button	The player will be hovering over buttons constantly as they decide what to click	The UIHover sound effect is played	Passed	
Valid	Not hovering the button	On the other hand, the player may also skip certain buttons, especially when the buttons do not take up the majority of the scene.	No sound effect will be played	Passed	
Valid	Clicking on the button	The player will always have to click a button to progress through the menus	The UISelect sound effect is played	Passed	
Boundary	Hovering over and then off the	Most of the time, the player will	The UIHover sound effect is played	Passed	

	button	hover over a button but not click it, and as such this needs to be tested correctly	once as the mouse hovers over, but not off		
--	--------	---	--	--	--

## Options

One of the settings that I am putting in the options menu are the volume controls for the master channel. This requires exposing the volume of the master channel in the mixer settings, and the name I have given it is masterVolume. This variable can be set via a slider on the UI, which will update this value when it is changed.

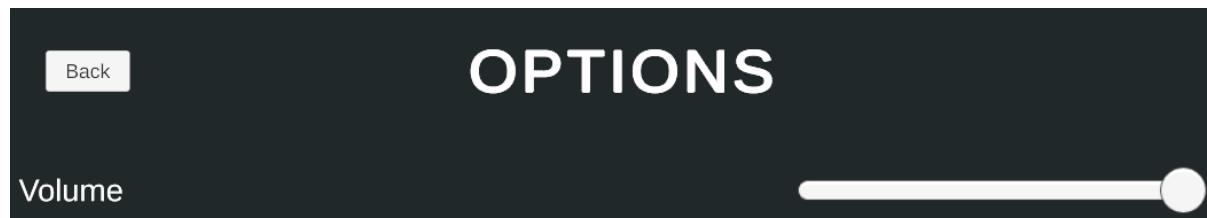


The code will be split into 2 parts, one in AudioManager, and another in Options, which is a script which calls the relevant external functions on other scripts. This menu will read the value of the slider, which goes between 0 and 1, and then use that to interpolate between -80 decibels (which is considered to be the mute option) and 0 decibels, which is the standard volume. This is then passed to SetMasterVolume() on AudioManager.

```
public void UpdateVolume()
{
    AudioManager.Instance.SetMasterVolume(Mathf.Lerp(-80, 0, volume.value)); // Set the master volume to the value of the slider
}
```

The manager only needs to set the exposed parameter using mixer.SetFloat(), with the supplied volume.

```
public void SetMasterVolume(float volume)
{
    // Set the volume of the Master channel
    mixer.SetFloat("masterVolume", volume);
}
```



*(The volume slider in the menu)*

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Not changing the slider	Most of the time, the player will not need to change the volume of the game when they open the menu, and as such this needs to be tested	The volume remains at its current value	Passed	
Valid	Changing the slider to any value	Typically, the player will need to set the volume to an intermediary volume for their needs	The volume will be adjusted according to the slider	Passed	
Boundary	Moving the slider to the minimum or the maximum	The player may need to mute the game, or return it to its original value, and this should be tested	The game's volume will be muted (if the slider is set to the minimum) or return to the normal value (if the slider is dragged to the maximum)	Passed	
Error	Dragging the slider out of its limits	The player will not attempt to drag the slider carefully, and will typically	The slider will stay at either the minimum or maximum	Passed	

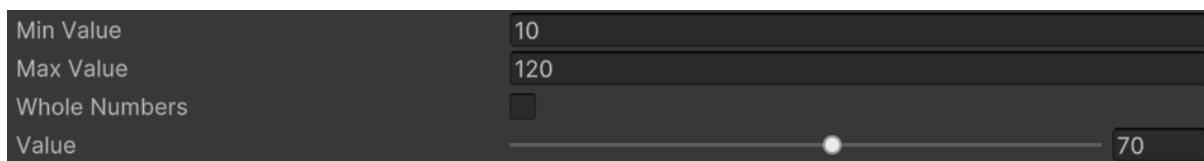
		drag it past its limits to set the minimum and maximum, and this should be tested appropriately			
--	--	---	--	--	--

The other setting that I will implement as an example is the ability to change the FOV when the player is moving around. This will also use a slider; however, I discovered that I could change the minimum and maximum value of the slider in the inspector. To decide what the minimum and maximum FOV should be, I consulted SillyAustralian:

#### *Interview*

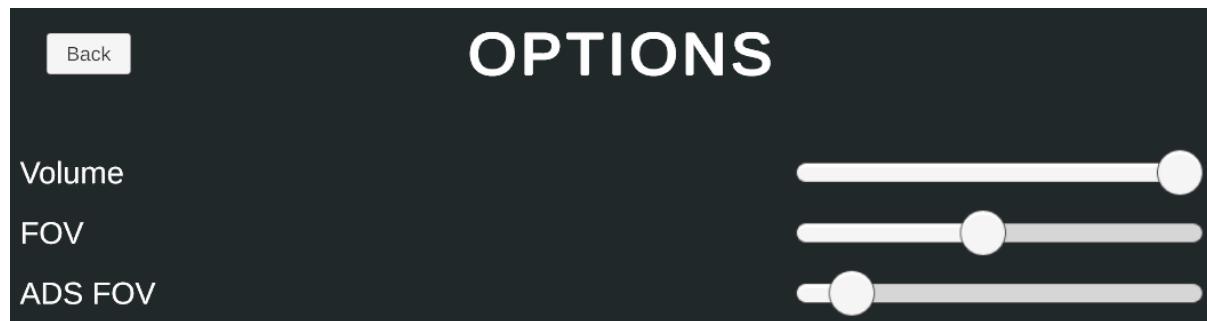
SillyAustralian: The typical values that games use for the camera's FOV range between 10 and 120, so do those as well so that players have a standard that they are comfortable with. As the default value, make this around 70. Finally, make sure the player can change the FOV for when they are zoomed in as well, and have this have the same range as the normal FOV. Call it ADS for Aim-Down-Sights.

These values can be put into the slider's settings:



When the sliders are changed, they will call a function in the Options menu, which will set the relevant values in the player's weapon interaction script, which have now been marked as public.

```
public void UpdateNormalFOV()
{
    PlayerManager.Instance.player.GetComponent<PlayerGunInteraction>().fov = fov.value; // Set the normal FOV
}
public void UpdateZoomedFOV()
{
    PlayerManager.Instance.player.GetComponent<PlayerGunInteraction>().zoomFov = zoomFov.value; // Set the zoomed in FOV
}
```



*(The final Options menu)*

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Not changing either slider	Most of the time, the player will not need to change their FOV for either option when they open the menu, and as such this needs to be tested	The FOV and ADS FOV remains at its current value	Passed	
Valid	Changing the FOV to any value	Typically, the player will need to set the FOV to an intermediary value for their needs	The FOV slider will be set to the correct value	Passed	
Valid	Changing the ADS FOV to any value	Typically, the player will need to set the ADS FOV to an intermediary value for their needs	The ADS FOV slider will be set to the correct value	Passed	
Boundary	Moving the either slider to the minimum or the maximum	While uncommon, some players may use extreme FOVs, and this should be	The appropriate slider will be set to either the minimum or the maximum	Passed	

		tested			
Error	Dragging either slider out of its limits	The player will not attempt to drag either slider carefully, and will typically drag it past its limits to set the minimum and maximum, and this should be tested correctly	The slider will stay at either the minimum or maximum	Passed	

## Movement

At this point, I can now implement the movement features that were of secondary priority: the special options. The first one I could implement is sliding, which first requires checking if the player is moving above the walking speed. If they are, and then they start crouching, then they enter a sliding state, which has lower friction and control. The player also gains a slight boost in the first second of sliding. To implement this boost, the acceleration function will be tweaked slightly. When they start sliding, the current time is recorded and each frame, the difference between the current time and the stored time is calculated. If this difference is below a certain value, then the acceleration value is higher. Otherwise, the acceleration value is extremely low. This limiting value needs to be decided by SillyAustralian:

### *Interview*

SillyAustralian: This initial boost should only be there for a very short amount of time. This amount of time should be a tenth of a second, as it ensures that anyone playing at the minimum acceptable frame rate of 30 fps will be able to experience the boost for a few frames.

Stakeholder Signature:



This feature can be put into code like below, where startedSliding is set upon the state change.

```
case PlayerMovementState.sliding:
    float currentTime = Time.time;
    float difference = currentTime - startedSliding;
    if (difference < 0.1)
    {
        acceleration = -5 * difference + 20;
    }
    else
    {
        acceleration = 0.5f;
    }
break;
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the crouch key at low speeds	This is input that the player will use, and the state needed to crouch, and hence needs to be tested appropriately	The player's height will be shrunk	Passed	
Valid	Pressing the crouch key at high speeds	This is input that the player will use, and the state needed to	The player's height will be shrunk, and they will gain a small bit of	Partial: the player gained infinite speed	Need to set a sliding cap

		slide, and hence needs to be tested appropriately	speed, before slowing down		
Boundary	Pressing the crouch key at high speeds and then again quickly	The player may use this to chain together a lot of speed and therefore should be tested corrected	The player's height will be shrunk, and they will gain speed before returning to the standing height.	Passed	
Erroneous	Pressing another key (e.g. M)	An input not related to the test should not affect the test	The player will remain at their standing height	Passed	

At the moment however, the player is able to gain an infinite amount of speed since there is no strict maximum sliding speed. The maximum sliding speed that the player can play depends on what speed they started sliding, since the player can enter this state at any point. Therefore, the maximum speed that the player can reach is not a set value. To combat this, the player will not experience a speed cap while they are still experiencing the initial boost, after which point the speed that they reach becomes the maximum speed that they can reach.

```

if (difference < 0.1)
{ // If its still initial
    acceleration = -5 * difference + 20; // Apply a high acceleration
    slideCapSet = false; // The maximum sliding limit hasn't been set
}
else
{
    acceleration = 0.5f; // Apply a low acceleration
    slideCapSet = true; // The maximum sliding speed has been set
}
break;
else if (surfaceVelocity.magnitude > maxWalkSpeed && isCrouched)
{ // If sliding
    movementState = PlayerMovementState.sliding;
    if (!wasSliding)
    { // The player has just changed state
        wasSliding = true; // Have started sliding
        startedSliding = Time.time; // Set the current time
    }
    if (slideCapSet)
    { // If the maximum sliding speed has been set
        maxSlidingSpeed = surfaceVelocity.magnitude; // Set the current velocity as the limit
    }
}

```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the crouch key at low speeds	This is input that the player will use, and the state needed to crouch, and hence needs to be tested appropriately	The player's height will be shrunk	Passed	
Valid	Pressing the crouch key at high speeds	This is input that the player will use, and the state needed to slide, and hence needs to be tested appropriately	The player's height will be shrunk, and they will gain a small bit of speed, before slowing down	Passed	
Boundary	Pressing the crouch key at high speeds and then again	The player may use this to chain together a lot of speed and	The player's height will be shrunk, and they will gain speed before	Passed	

	quickly	therefore should be tested corrected	returning to the standing height.		
Erroneous	Pressing another key (e.g. M)	An input not related to the test should not affect the test	The player will remain at their standing height	Passed	

I also noted that the same issue exists with boosting, as past a certain point, the amount of speed gained is not able to be counteracted with the friction that they experience. To calculate the maximum speed that the player can reach when boosting, I can add together the initial velocity, and the total velocity gained from the boost. The initial velocity can then be stored to add an extra check into deciding whether the player is walking or not, since if the player starts boosting below the walking or sprinting speed, the game should not automatically transfer them to that state.

```
private void Boost(InputAction.CallbackContext inputType)
{
    movementState = PlayerMovementState.boosting; // Set state to boosting
    preBoostVelocity = surfaceVelocity.magnitude < maxSprintSpeed ? maxSprintSpeed : surfaceVelocity.magnitude; // If the player is moving below the maximum sprint speed, set the calculated speed to the sprinting speed, otherwise set it to the current speed
    inputDirection = playerInputs.Player.Movement.ReadValue<Vector2>().normalized; // Get the player's inputs

    Vector3 boostMovement = player.rotation * new Vector3(inputDirection.x, 0, inputDirection.y) * preBoostVelocity * (boostMultiplier - 1); // Work out how much extra speed the player will gain
    maxBoostSpeed = velocity.magnitude + boostMovement.magnitude; // Sets the maximum speed
    player.AddForce(boostMovement, ForceMode.VelocityChange); // Apply force
}
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the boost key	This should be the only key used for boosting, and hence should respond appropriately	The player's speed increases by a huge amount, and then the speed cap is set	Passed	
Erroneous	Pressing another key (e.g. M)	Keys not used for boosting should not activate it	The player will remain at their current speed	Passed	

Then next special movement technique is double jump, which requires 2 more variables: airJumpsLeft and airJumpsTotal. airJumpsLeft will track how many more times the player can jump in mid-air, while the airJumpsTotal will be used to reset airJumpsLeft. Deciding whether to use a double jump or not requires the use of an if statement, where the condition is based on if the player is on the ground or not. If they are, they can jump normally, but if they aren't, then they consume a double jump. This

also means that an extra check is required before that if the player is moving downwards, they must gain enough vertical speed to completely counteract this downwards movement to make sure that the double jump is effective.

```
if (velocity.y < 0)
{
    // If the player is moving downwards, provide enough force to move them back upwards
    direction.y -= velocity.y;
}
if (isGrounded)
{
    // If on the ground
    Debug.Log("Jumped");
    player.AddForce(direction, ForceMode.VelocityChange);
}
else if (airJumpsLeft > 0)
{
    // If allowed to double jump
    airJumpsLeft--; // Remove 1 double jump
    Debug.Log($"Air jumped, jumps left: {airJumpsLeft}");
    player.AddForce(direction, ForceMode.VelocityChange);
}
```

When the player touches something, they regain all of their air jumps, encouraging them to make contact with a wall or the ground at some point.

```
foreach (ContactPoint contact in collision.contacts)
{
    float slopeAngle = Vector3.Angle(contact.normal, Vector3.up); // Calculate the angle of the slope from the vertical
    airJumpsLeft = airJumpsTotal; // Resets the airjumps
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the jump key on the ground	As the only input being used to jump, it is important that it works, and that it has not been affected by the double jump	The character will move upwards, then move back down, and no air jumps are used	Passed	
Valid	Pressing the jump key in the air while moving upwards	This tests the player's ability to double jump	The player will move upwards faster, using an air jump, and will fall back down eventually	Passed	
Valid	The player pressing the	Since the player is likely	The player will continue	Passed	

	jump key while in mid-air when they have no air jumps left	to have used all their double jumps, they should no longer be able to jump in mid air	accelerating downwards		
Boundary	Touching a surface when the player has no air jumps left	The player will usually exhaust all their double jumps, and they need to be able to get them back	airJumpsLeft will be set to airJumpsTotal	Passed	
Boundary	Pressing the jump key in the air while moving downwards	The player should still be able to get the full use out of their double jump without it being a waste	The player will move upwards, using an air jump, then move down	Passed	
Erroneous	Pressing another key (e.g. M)	Since there is only one input, nothing else should cause the player to jump, and if there is, that means something is significantly broken	The player will remain still	Passed	

As I was looking through the code and reading some of Unity's forums, I discovered that it was incorrect to use `Rigidbody.AddForce()` outside of `FixedUpdate()` since it is something that works with the physics system. This can cause errors and issues when the framerate does not match the tick rate of the physics engine (this is set to 50 ticks per second by default) and the usual player will normally be running the game at frame rates that almost never match exactly 50 fps. Therefore, I created a property called movement. This stores the total movement that all the functions want to apply to the player's body, weighted based upon the time it took for that frame. This is then applied each time `FixedUpdate()` is applied and then reset to 0. All functions will exist within

Update(), which runs every frame, and record the player's movement. The exemption to this is gravity, which can run every physics tick instead, since the player's input does not affect it.

```
Vector3 direction = target * maxSpeed - surfaceVelocity; // Work out how much extra acceleration is needed
float directionMag = direction.magnitude; // Temporarily store the magnitude of the direction
direction = Vector3.ProjectOnPlane(direction, groundNormal).normalized * directionMag; // Project direction on the ground and maintain its magnitude
Vector3 targetMovement = direction.normalized * acceleration; // Work out the next movement
targetMovement -= targetMovement * FrictionMultiplier(); // take a percentage away from friction
movement += targetMovement * Time.deltaTime; // add it to the movement for the next physics timestep
```

```
private void FixedUpdate()
{
    Gravity(); // Apply gravity
    player.AddForce(movement, ForceMode.VelocityChange); // Apply total movement
    movement = Vector3.zero; // Reset movement
}
```

Another optimisation I noticed was the number of variables in the movement function. Originally, to calculate the amount of movement required, 3 variables are used to store the movement at different stages. However, all 3 are not needed, since the difference in the values do not do anything. Instead, only 1 variable is used, and I was able to rewrite these lines of code to use only target instead of target, targetMovement, and direction. This reduces the amount of memory the game uses, reducing the system requirements.

```
target = target * maxSpeed - surfaceVelocity; // Work out how much extra acceleration is needed
target = Vector3.ProjectOnPlane(target, groundNormal).normalized; // Project target on the ground
target *= acceleration; // Work out the next movement
target -= target * FrictionMultiplier(); // take a percentage away from friction
movement += target * Time.deltaTime; // add it to the movement for the next physics timestep
```

Aside from these optimisations, I also moved onto implementing the ability to run on walls. Wallrunning will use the same sort of underlying math as the rest of the movement functions, but it will be slightly tweaked to move on a wall. Firstly, CollisionDetected() needs to be edited to include a few more variables, which allow it to decide whether the player is on the floor or on a wall. This is done by iterating through every contact point and working out the angle, and if this angle is not too steep, then the player is on the ground, and the function decides that the player is grounded and ends there, storing the normal of the ground for use within Movement(). However, if there is a point that is too steep, then the function marks that the player is touching a wall object. If the player is not grounded after every point, then the player is stuck on the wall, and they are set to be wallrunning, with the normal of the wall being stored for later use.

```

public void CollisionDetected(Collision collision)
{ // This function is called externally by the body
    if (collision.contacts.Length > 0)
    { // If the player is touching something
        bool onWall = false; // Track if the player is on a wall
        isGrounded = false; // Track if the player is on a floor
        foreach (ContactPoint contact in collision.contacts)
        {
            float slopeAngle = Vector3.Angle(contact.normal, Vector3.up); // Calculate the angle of the slope from the vertical

            airJumpsLeft = airJumpsTotal; // Resets the airjumps
            if (slopeAngle <= maxSlope)
            { // If on the ground
                isGrounded = true;
                groundNormal = contact.normal;
                break;
            }
            else
            { // On the wall
                onWall = true;
                wallNormal = contact.normal;
            }
        }
        if (onWall && !isGrounded)
        { // The player is wallrunning
            movementState = PlayerMovementState.wallrunning;
            stuckToWall = true;
        }
    }
    else
    { // Not touching anything
        isGrounded = false; // In the air
        stuckToWall = false; // No longer on the wall
        movementState = PlayerMovementState.walking; // apply normal walking state
        groundNormal = Vector3.up; // Reset groundNormal to default
    }
}

```

As mentioned, Wallrun() will contain similar code to Movement(), running some extra checks first beforehand, and using the normal of the wall instead of the floor. These checks include if the player is attempting to move away from the wall, and if the player is moving far too slow on the wall. In either case, the player is kicked off the wall with an extra boost of speed. This makes sure that the player doesn't accidentally reattach to the same wall and make them unable to move into any other state. The player is then set to the walking state as a default state.

```

private void Wallrun()
{
    Vector3 target = player.rotation * new Vector3(inputDirection.x, 0, inputDirection.y); // Get the intended movement globally
    if (Mathf.Abs(Vector3.Angle(wallNormal, target)) < wallSeparationAngle)
    { // If attempting to move away from the wall
        stuckToWall = false; // Remove from wall
        movement += wallNormal * separationBoost * Time.deltaTime; // provide a boost to be pushed off the wall
        movementState = PlayerMovementState.walking; // Set state to walking
        return;
    }
    else if (player.linearVelocity.magnitude < wallFallSpeed)
    {
        stuckToWall = false; // Remove from wall
        movement += wallNormal * separationBoost * Time.deltaTime * 0.1f; // Provide a small boost to be kicked off the wall
        movementState = PlayerMovementState.walking; // Set state to walking
        return;
    }

    float maxSpeed = MaxSpeed(); // Get the maximum speed for that state
    float acceleration = CalculateAccelerationMultiplier(); // Get the acceleration for that state
    Vector3 wallVelocity = Vector3.ProjectOnPlane(player.linearVelocity, wallNormal);

    if (wallVelocity.magnitude > maxSpeed)
    { // If moving above the maximum wallrunning speed
        acceleration *= wallVelocity.magnitude / maxSpeed;
    }
    target = target * maxSpeed - wallVelocity; // Work out how much extra acceleration is needed
    target = Vector3.ProjectOnPlane(target, wallNormal).normalized; // Project target on the ground
    target *= acceleration; // Work out the next movement
    target -= target * FrictionMultiplier(); // take a percentage away from friction
    movement += target * Time.deltaTime; // add it to the movement for the next physics timestep
}

```

The player will also have lower gravity while on the wall, and this will need to be taken into account in the gravity function.

```

private void Gravity()
{
    if (stuckToWall)
    { // If wallrunning
        movement += Physics.gravity / -9.81f * gravity * gravityMultiplier * Time.fixedDeltaTime;
    }
    else if (!isGrounded)
    { // If in the air
        movement += Physics.gravity / -9.81f * gravity * Time.fixedDeltaTime;
    }
}

```

Finally, when on the wall, pressing the jump input should allow the player to be kicked away from the wall, with a boost parallel to the wall normal.

```

Vector3 direction = Vector3.up * jumpForce;
if (movementState == PlayerMovementState.wallrunning)
{ // If wallrunning
    direction += wallNormal * jumpForce; // Make sure to push the player away from the wall
    direction = direction.normalized * jumpForce; // Push with a predetermined magnitude
    if (velocity.y < 0)
    { // If moving down
        direction.y -= velocity.y; // Move the player back upwards
    }
    movement += direction; // Add force for the next tick
    return;
}

```

Type	What data fits	Justification	Expected	Actual	Comments
------	----------------	---------------	----------	--------	----------

	criteria				
Valid	Pressing forward on the wall	This is the most likely action that the player is going to do when they are on the wall, and as such, this should be the highest priority test	The player will move forward on the wall	Passed	
Valid	Facing perpendicular to the wall	The player may want to dismount from the wall	The player will gain a speed boost as they disconnect from the wall and enter a walking state.	Passed	
Valid	The player pressing the jump key while on the wall	Since the aim of the game is to maintain speed, the player will want to jump away from the wall instead of the other methods to maximise the speed gain	The player will dismount from the wall and gain some speed in the direction of the wall normal and upwards	Passed	
Boundary	Slowing down on the wall below a threshold	The player may attempt to turn around on the wall, or slow down, and it makes sense for this to mean that they can no longer hold onto the wall.	The player is pushed away from the wall	Passed	
Boundary	Pressing the jump key in the air while	The player is still affected by gravity, if	The player will dismount	Passed	

	moving downwards on the wall	only by a little bit, so the player can end up moving downwards, which may make the jump useless. Hence this needs to be counteracted	from the wall and gain some speed in the direction of the wall normal and upwards		
Erroneous	Pressing another key (e.g. M)	An input not related to the system should not affect it	The player will slow down on the wall due to friction	Passed	

## End of prototype tests

### UI Sounds

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Hovering over the button	The player will be hovering over buttons constantly as they decide what to click	The UIHover sound effect is played	Passed	
Valid	Not hovering the button	On the other hand, the player may also skip certain buttons, especially when the buttons do not take up the majority of the scene.	No sound effect will be played	Passed	
Valid	Clicking on the button	The player will always have to click a button to progress	The UISelect sound effect is played	Passed	

		through the menus			
Boundary	Hovering over and then off the button	Most of the time, the player will hover over a button but not click it, and as such this needs to be tested correctly	The UIHover sound effect is played once as the mouse hovers over, but not off	Passed	

*Volume Slider*

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Not changing the slider	Most of the time, the player will not need to change the volume of the game when they open the menu, and as such this needs to be tested	The volume remains at its current value	Passed	
Valid	Changing the slider to any value	Typically, the player will need to set the volume to an intermediary volume for their needs	The volume will be adjusted according to the slider	Passed	
Boundary	Moving the slider to the minimum or the maximum	The player may need to mute the game, or return it to its original value, and this should be tested	The game's volume will be muted (if the slider is set to the minimum) or return to the normal value (if the slider is dragged to	Passed	

			the maximum)		
Error	Dragging the slider out of its limits	The player will not attempt to drag the slider carefully, and will typically drag it past its limits to set the minimum and maximum, and this should be tested appropriately	The slider will stay at either the minimum or maximum	Passed	

*FOV Slider*

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Not changing either slider	Most of the time, the player will not need to change their FOV for either option when they open the menu, and as such this needs to be tested	The FOV and ADS FOV remains at its current value	Passed	
Valid	Changing the FOV to any value	Typically, the player will need to set the FOV to an intermediary value for their needs	The FOV slider will be set to the correct value	Passed	
Valid	Changing the ADS FOV to any value	Typically, the player will need to set the ADS FOV to an intermediary value for their needs	The ADS FOV slider will be set to the correct value	Passed	

		needs			
Boundary	Moving the either slider to the minimum or the maximum	While uncommon, some players may use extreme FOVs, and this should be tested	The appropriate slider will be set to either the minimum or the maximum	Passed	
Error	Dragging either slider out of its limits	The player will not attempt to drag either slider carefully, and will typically drag it past its limits to set the minimum and maximum, and this should be tested correctly	The slider will stay at either the minimum or maximum	Passed	

*Sliding*

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the crouch key at low speeds	This is input that the player will use, and the state needed to crouch, and hence needs to be tested appropriately	The player's height will be shrunk	Passed	
Valid	Pressing the crouch key at high speeds	This is input that the player will use, and the state needed to slide, and hence needs to be tested appropriately	The player's height will be shrunk, and they will gain a small bit of speed, before slowing down	Passed	

Boundary	Pressing the crouch key at high speeds and then again quickly	The player may use this to chain together a lot of speed and therefore should be tested corrected	The player's height will be shrunk, and they will gain speed before returning to the standing height.	Passed	
Erroneous	Pressing another key (e.g. M)	An input not related to the test should not affect the test	The player will remain at their standing height	Passed	

*Boosting*

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the boost key	This should be the only key used for boosting, and hence should respond appropriately	The player's speed increases by a huge amount, and then the speed cap is set	Passed	
Erroneous	Pressing another key (e.g. M)	Keys not used for boosting should not activate it	The player will remain at their current speed	Passed	

*Double Jumping*

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing the jump key on the ground	As the only input being used to jump, it is important that it works, and that it has not been affected by the double jump	The character will move upwards, then move back down, and no air jumps are used	Passed	
Valid	Pressing the	This tests the	The player	Passed	

	jump key in the air while moving upwards	player's ability to double jump	will move upwards faster, using an air jump, and will fall back down eventually		
Valid	The player pressing the jump key while in mid-air when they have no air jumps left	Since the player is likely to have used all their double jumps, they should no longer be able to jump in mid air	The player will continue accelerating downwards	Passed	
Boundary	Touching a surface when the player has no air jumps left	The player will usually exhaust all their double jumps, and they need to be able to get them back	airJumpsLeft will be set to airJumpsTotal	Passed	
Boundary	Pressing the jump key in the air while moving downwards	The player should still be able to get the full use out of their double jump without it being a waste	The player will move upwards, using an air jump, then move down	Passed	
Erroneous	Pressing another key (e.g. M)	Since there is only one input, nothing else should cause the player to jump, and if there is, that means something is significantly broken	The player will remain still	Passed	

*Wallrunning*

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Pressing forward on the wall	This is the most likely action that the player is going to do when they are on the wall, and as such, this should be the highest priority test	The player will move forward on the wall	Passed	
Valid	Facing perpendicular to the wall	The player may want to dismount from the wall	The player will gain a speed boost as they disconnect from the wall and enter a walking state.	Passed	
Valid	The player pressing the jump key while on the wall	Since the aim of the game is to maintain speed, the player will want to jump away from the wall instead of the other methods to maximise the speed gain	The player will dismount from the wall and gain some speed in the direction of the wall normal and upwards	Passed	
Boundary	Slowing down on the wall below a threshold	The player may attempt to turn around on the wall, or slow down, and it makes sense for this to mean that they can no longer hold onto the wall.	The player is pushed away from the wall	Passed	
Boundary	Pressing the	The player is	The player	Passed	

	jump key in the air while moving downwards on the wall	still affected by gravity, if only by a little bit, so the player can end up moving downwards, which may make the jump useless. Hence this needs to be counteracted	will dismount from the wall and gain some speed in the direction of the wall normal and upwards		
Erroneous	Pressing another key (e.g. M)	An input not related to the system should not affect it	The player will slow down on the wall due to friction	Passed	

## Interview

SillyAustralian: All of these systems are working very well! However, do note that there is very little time remaining to the deadline, and moving forward, we would only like you to have the single line interpretation of the dialogue system fully implemented, so that we can have a minimum working demo of this for us to build upon. There is unlikely to be any extra time for any other features so you can leave it at that.

Stakeholder Signature:



## Prototype 5

### Dialogue

Implementing the dialogue system to fulfil all its requirements was incredibly difficult and arduous. To be able to easily add multiple languages, the best solution to this would be to have every single dialogue line in a database, where the different fields are the different languages. All one would need was the record ID and the related language, and the correct line of dialogue would be loaded.

The interactions will be stored in external files with the extension of .dlg, where the flow of lines will be marked with special syntax, of which will be stored in 2 locations: some within the line data in the database, and others within the .dlg file itself. Within the line itself, the syntax is:

- \_ which denotes the text surrounded by them to be in italics
- \* which denotes the text surrounded by them to be in bold
- < and > surround colour information about the text written in hexadecimal (#FFFFFF) like “<#FFFFFF example> where “example” will be in white.
- \ is the escape character, allowing any of the above characters to be displayed, so long as it is preceded by it

For the .dlg file, the syntax varies more, since rather than styling the text, it instead defines the flow of the lines:

- Dialog sections are marked with [], which is how the dialog will be able to jump, and change based on variables. Dialog sections won't naturally progress to each other unless a -> is used to go to the next section, allowing the last section to continue forever.
- Choices for the player are marked with | at the beginning, and the first choice is used as the title of the choice. These set a local ‘choice’ variable to the number of the choice, which can then be checked and jumped to after. After each choice, a comma is there to indicate another choice is there, except for the last one, signifying it is the last choice.
- Variables will be how the dialog will react to external events. To mark a variable-based line, it will be preceded by @. This will prompt the game to check for that relevant value being set in the save file. If the condition is fulfilled, then the dialog will go to the dialog section that is labelled after the -> that succeeds the condition. Otherwise, it will go to the next line. Chaining together many of these will allow an if-else-like behaviour. If a variable is on a line by itself, it can be assigned to a value. Variables can also be assigned values similarly, instead using '=' to set the value, and no -> will be required after.

- To request each line of dialog, the line is prefaced with an actor ID, followed by :: then the line ID. The actor IDs are labelled with #Ax and the line IDs are labelled with #Lx, where the x in both is the record ID in the database for each of them.

As you might have noticed, this would require me to create a custom parser for all of this, along with managers for the subtitles and databases. Using similar code to the databases above, I could implement this. I would need two databases, one for storing the entire list of actors, and another for storing the dialogue lines. Having all the actors in a database would make the parser easier to construct, since it would not need to be aware beforehand of which scene and interaction it was in – it could be easily generalised to handle all situations. A separate manager would be responsible for assigning the actors to the actual characters in the scene, but this would be included in the scene data making this a worthwhile option. I first started off by making a way to create the database and table connection, resulting in the below code:

```
private IDbConnection OpenDb(){
    string database = "URI=file:" + Application.streamingAssetsPath + "/" + "Dialogue.sqlite";
    IDbConnection connection = new SqliteConnection(database);
    connection.Open();
    IDbCommand createTable = connection.CreateCommand();

    createTable.CommandText = $"CREATE TABLE IF NOT EXISTS Dialogue (id INTEGER PRIMARY KEY, english TEXT)";
    createTable.ExecuteReader();

    return connection;
}
```

This code makes a reference to the StreamingAssets folder of Unity, which is how developers are able to include files unchanged with the build of their application. Usually when a file is included, it is serialized to make it easier and faster to load into memory. This also means that the format of the file changes, so it may no longer be possible to access it like you may have before. The way around this is StreamingAssets, which is where all the databases and .dlg files will exist. This version of the code does not take into account the different ways that different OS define paths to files, nor does it take into account that multiple different databases exist. To handle the different databases, and the different names for the columns, the function was changed to have two parameters: a string databaseName, and an array of strings that contained the columns. To handle the array, a string had to be made that combined the whole array into 1 string.

```

private IDbConnection OpenDb(string databaseName, string[] columns){
    string database = "URI=file:" + Application.streamingAssetsPath + "/" + "Dialogue.sqlite";
    IDbConnection connection = new SqliteConnection(database);
    connection.Open();
    IDbCommand createTable = connection.CreateCommand();

    string columnText = "";
    foreach(string column in columns){
        columnText += column + ", ";
    }
    columnText = columnText.Substring(0, columnText.Length-2);
    createTable.CommandText = $"CREATE TABLE IF NOT EXISTS {databaseName} (id INTEGER PRIMARY KEY, {columnText})";
    createTable.ExecuteReader();

    return connection;
}

```

The for loop would iterate over each string in the array, and add it to the already combined total, along with a comma and a space to separate the different fields. The last field does not need a comma after it, so the last 2 characters will be cut off by taking a substring from the start of columnText, to 2 before the end of it.

To then fix the issue of the different OS, instead of concatenating strings together, I could use the Path class from System.IO which has a function that automatically stiches together a path with the appropriate connectors. I also discovered that C# had range slicing for substrings like Python did, and the code was far shorter to use, and as such I would be using the range operator going forward. To simplify the access of opening the actor or dialogue database to external classes, 2 public functions were made to fill in the required data into the OpenDb function.

```

using UnityEngine;
using System.Data;
using Mono.Data.Sqlite;
using UnityEngine.UIElements;
using System.IO;

public class DialogueDatabaseManager : MonoBehaviour
{
    public IDbConnection OpenDialogueDb()
    {
        return OpenDb("Dialogue", new string[] {"english TEXT"});
    }

    public IDbConnection OpenActorDb()
    {
        return OpenDb("Actor", new string[] {"actor TEXT"});
    }

    private IDbConnection OpenDb(string databaseName, string[] columns){
        string database = "URI=file:" + Path.Combine(Application.streamingAssetsPath, databaseName + ".sqlite");
        IDbConnection connection = new SqliteConnection(database);
        connection.Open();
        IDbCommand createTable = connection.CreateCommand();

        string columnText = "";
        foreach(string column in columns){
            columnText += column + ", ";
        }
        columnText = columnText[..^2];
        createTable.CommandText = $"CREATE TABLE IF NOT EXISTS {databaseName} (id INTEGER PRIMARY KEY, {columnText})";
        createTable.ExecuteReader();

        return connection;
    }
}

```

Now that other scripts could easily setup a connection with the dialogue or actor database, it was time to start implementing the .dlg files. When the program had to go through the file, it first needed to preprocess it and find where the dialog sections were on the lines of the file. This also needed me to be able to read the text within the file.

Unity has a built-in asset type called TextAsset, and the useful property of it was that the text could easily be extracted from within it with a single property. Writing code to sift through the lines and add it to a list of the line number and dialog section name was not terribly difficult.

```
private void PreprocessSections(){
    string[] fileLines = dialogueFile.text.Split('\n');
    for (int i = 0; i < fileLines.Length; i++){
        string line = fileLines[i];
        if (line.StartsWith("[") && line.EndsWith("]")){
            dialogueSections.Add((i, line.Substring(1, line.Length - 2)));
        }
    }
}
```

It consists of just a simple for loop, checking whether the end and start of a line had square brackets, the defining feature of dialog section headers. The code was extremely simple, but only if the .dlg file was to be recognised as a TextAsset. Unity unfortunately, was unable to do so, even with many attempts, including attempting to write a ScriptedImporter for the filetype.

```
using UnityEngine;
using UnityEditor.AssetImporters;
using System.IO;

You, yesterday | 1 author (You)
[ScriptedImporter(1, "dlg")]
0 references
public class DLGImporter : ScriptedImporter
{
    0 references
    public override void OnImportAsset(AssetImportContext context){
        TextAsset asset = new TextAsset(File.ReadAllText(context.assetPath));
        context.AddObjectToAsset("dlg", asset, Resources.Load("dlg icon.png") as Texture2D);
        context.SetMainObject(asset);
    }
}
```

This was standard code to do the following, albeit without the icon (this was added as an easier check to see if the file was imported correctly), but unfortunately it would not work. Even after scouring for a solution to this problem, none were found, and as such I had to go for the less memory-efficient and slightly worse approach of using the C# method to open and reading file contents – StreamReader. While the File class exists, it

turns out to be far slower to StreamReader on its own, and as such I went with the below changed code to accommodate StreamReader.

```
public class DialogueInteraction : MonoBehaviour
{
    1 reference
    public GameObject gameManager;
    3 references
    private DialogueDatabaseManager ddm;
    0 references
    public GameObject dialogueManager;
    3 references
    private IDbConnection dialogueConnection;
    2 references
    private IDbConnection actorConnection;
    0 references
    private List<int, string> actorsInInteraction = new List<int, string>();
    1 reference
    public TextAsset dialogueFile;
    2 references
    private string filepath;
    1 reference
    [SerializeField] private List<int, string> dialogueSections = new List<int, string>();
    1 reference
    [SerializeField] private int nextLine = 1;
    0 references
    void Start()
    {
        ddm = gameManager.GetComponent<DialogueDatabaseManager>();
        filepath = Path.Combine(Application.streamingAssetsPath, "Dialogue", dialogueFile.name + ".dlg");
        PreprocessSections();
    }
    1 reference
    private void PreprocessSections(){
        StreamReader reader = new StreamReader(filepath);
        int lineNumber = 1;
        while(!reader.EndOfStream){
            string line = reader.ReadLine();
            if(line.StartsWith("[") && line.EndsWith("]")){
                dialogueSections.Add((lineNumber, line[1..^1]));
            }
            lineNumber++;
        }
    }
}
```

The new code is very similar to the previous, but has a custom counter instead, and reads off a new line using StreamReader.ReadLine(). It stops when the reader detects it has reached the end of the file.

The next part of this system that I will implement is the ability to convert between the syntax of the dialogue line in the database, and the Rich Text format used by Unity in text objects on the UI. This can be done by iterating through every character in the string and replacing certain characters with the appropriate alternate version. If the character it comes across is not part of the registered syntax, then it skips over it. Similarly, if it comes across a '\', then it will skip over the next character as well.

While this works for the closing tag of the colour markings, a little bit more effort will be needed for the starting colour tag and the italics and bold markings. When reading the starting colour tag, the transpiler will cut out a substring of the line to read the following hex code of the colour. It is important that `++i` is used instead of `i++` to skip over the hashtag as the pre-increment shortcut and the post-increment shortcut differ in the values returned. `i++` will return the current value and then increment it after the compiler has moved onto the next line (when it has passed a semi-colon), while `++i` will

return the current value incremented, not waiting for the next semi-colon. While this seems like they would have performance differences, the C# compiler when allowed to perform optimisations will optimise them to have the same run-time, leading to no performance decreases from using one or another. The substring can then be put into <color=> to form the final tag used by the Rich Text system, and the length of the hex code is skipped.

To handle bolding and italicising, the transpiler will keep track of whether it has yet to reach the first italic or bold symbol out of the pair. If it reaches one, it puts in the appropriate opening tag and inverts the tracking variable. If it has already come across one and it comes across another, then it will place the opposing tag, and inverts that tracking variable.

```
private string ConvertLine(string line){
    string copy = "";
    bool firstItalics = true; // Tracks whether the first underscore is yet to come
    bool firstBold = true; // Tracks whether the first asterisk is yet to come
    for (int i = 0; i < line.Length; i++){ // For every character in the line
        if (line[i] == '\\\\'){ // Ignore the next character
            i++; // Skip over the next character
        } else if (line[i] == '<'){
            string hexColour = line.Substring(++i, 7); // Grab a colour code
            copy += "<color=" + hexColour + ">"; // Add the hexcode along with the <color= >
            i += 6; // Skip over the hexcode
        } else if (line[i] == '>'){
            copy += "</color>"; // Replace with the ending tag
        } else if (line[i] == '_'){
            if (firstItalics){
                copy += "<i>"; // Replace with the starting italics tag
                firstItalics = false;
            } else{
                copy += "</i>"; // Replace with the ending italics tag
                firstItalics = true;
            }
        } else if (line[i] == '*'){
            if (firstBold)
            {
                copy += "<b>"; // Replace with the starting bold tag
                firstBold = false;
            } else{
                copy += "</b>"; // Replace with the ending bold tag
                firstBold = true;
            }
        } else{
            copy += line[i]; // Just copy the character
        }
    }
    return copy;
}
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Reading	The majority	The syntax	Passed	

	normal syntax	of the transpiler's job is to deal with the syntax, and this makes sense as a valid test	will be replaced with the appropriate value		
Valid	Reading a backslash	The transpiler must also be able to deal with an escaped character	The following character is skipped over	Passed	
Boundary	Reading a character that is not part of the syntax	The majority of the line will be like this, but the transpiler should not interpret it as syntax	The character is passed through the system	Passed	

[19:48:33] This is a *spooky* line filled with **great** things ---and **bad** things.  
UnityEngine.Debug:Log (object)

(The testing output successfully formatted as Rich Text.)

To move onto the tokenizer, the first thing that must be done is design a tree node data structure. Tokenizers will have a tree that allows them to navigate each token appropriately, since some tokens may have their own children. The tree node needs to have a list of its children, along with a dictionary of values assigned to it. It also has a simple method to add a type, which will be used constantly by the tokenizer, along with a method to retrieve the type.

```
using System.Collections.Generic;

public class TreeNode
{
    public Dictionary<string, string> value = new Dictionary<string, string>();
    private List<TreeNode> children = new List<TreeNode>();
    public void AddChild(TreeNode child){
        children.Add(child);
    }
    public List<TreeNode> GetChildren(){
        return children;
    }
    public void AddType(string typeValue){
        value.Add("type", typeValue);
    }
    public string GetNodeType(){
        return value["type"];
    }
}
```

The tokenizer begins by checking if the file reader is null, and if it is, it makes a new StreamReader for the .dlg file. It then needs to skip over the lines that have already been read in the file, as it keeps a track of what line the interaction is on. Once it has skipped over the lines, it creates a new TreeNode called fileTree, which is necessary if the current line being read is a choice node, since multiple lines will need to be read instead of 1. Then so long as the file has not reached its end, the tokenizer will take the current line, convert it into a byte array so it can be used inside a MemoryStream object, which is then passed into another StreamReader. It checks whether the line's length is below 3, since if line only has 3 characters, then there is no way it has enough information to be an entire line. This is because the minimum length a line can be comes about with the jump syntax which is 2 characters long. However, it also needs a section header afterwards, which is required to be at least 1 character long. Therefore, the minimum line length is 3. After this check, it makes a new TreeNode called lineTree, which tracks the tokens in a line, but does not inherently carry any information itself, and then adds this as a child of fileTree. The tokenizer will then create tokens from the line, after which it will then check if the last node has a type of 'choice'. If so, then if this node does not also have 'end' in its dictionary, then the tokenizer must not return and continue onto the next line. It will then read the next line, checking that the next line both exists and is not a dialogue section header. If this is true, then the line number is incremented, and the current line is set to the line that was just read. If the node was not a choice node, the tokenizer then exists the loop and returns the fileTree.

```

private TreeNode Tokenizer(){
    if (fileReader == null){
        fileReader = new StreamReader(filepath); // Reopen the file
    }
    for (int i = 0; i < lineNumber; i++){
    {
        fileReader.ReadLine(); // Skip Lines until the current one is reached
    }
    TreeNode fileTree = new TreeNode(); // This is here in case a choice exists, in which case multiple lines need to be read at once
    int lastlineNumber = lineNumber - 1;
    string line = fileReader.ReadLine(); // Read the current Line
    while (!fileReader.EndOfStream && lastLineNumber != lineNumber){
        byte[] lineBytes = Encoding.UTF8.GetBytes(line); // Convert into a list of characters
        MemoryStream stream = new MemoryStream(lineBytes); // Make into a stream for StreamReader later
        lastLineNumber = lineNumber; // Set the previous Line
        if (line.Length < 3){ // The minimum length of the line cannot go below 3
            throw new System.Exception($"Tokenizer: The line '{line}' is missing information.");
        }
        StreamReader lineReader = new StreamReader(stream); // Prepare for reading per character
        TreeNode lineTree = new TreeNode(); // New node for the line
        fileTree.AddChild(lineTree); // Add the node to the file tree
        ReadNext(lineReader, lineTree); // Tokenize the line
        bool shouldStay = false; // Initialise shouldStay
        if (filetree.GetChildren()[1].GetNodeType() == "choice"){ // If the node is a choice node
            if (filetree.GetChildren()[1].value["end"] == "false"){ // If it is not the last node in the choice - it is necessary to put this on a separate line to avoid an error
                shouldStay = true; // Require reading the next line
            }
        }
    }

    string nextline = fileReader.ReadLine(); // Check the next line
    if (nextline[0] != '[' && nextline != null){ // If the next line isn't a dialogue section header or the end of the file
        lineNumber++; // Now on the next line
        line = nextline; // Set the current line
        if (!shouldStay){ // If staying is not required
            break; // Exit
        }
    }
}
return fileTree; // Return the read lines
}

```

ReadNext() is essentially a dispatcher function, reading the next character in the line and deciding based on the character, it will decide what relevant function it needs to call to make a new token. It reads a new character using ReadChar(), which internally skips over spaces using the ASCII code of the character. Once a non-space character has been reached, it will attempt to cast it into a character, and it will return it. This might fail if there are no more characters left to read, in which case the read character will be equal to -1. If this is the case, it will return an error.

```

private void ReadNext(StreamReader line, TreeNode lineTree)
{
    while (!line.EndOfStream)
    { // While the end of the line hasn't been reached
        char token = ReadChar(line);
        if (token == '#')
        {
            ReadID(line, lineTree);
        }
        else if (token == '@')
        {
            ReadIf(line, lineTree);
        }
        else if (token == '-')
        {
            ReadJump(line, lineTree);
        }
        else if (token == '|')
        {
            ReadChoice(line, lineTree);
        }
        else if (token == ':')
        {
            token = ReadChar(line);
            if (token == ':')
            {
                ReadSeparator(line, lineTree);
            }
            else
            {
                throw new Exception($"Tokenizer: The starting character '{token}' was not recognised.");
            }
        }
    }
}

```

One of the functions it dispatches is ReadID(), which is called when the read character is a hashtag. It reads of an extra character and then knows that the next characters will be a number, which it dispatches ReadNumber() to handle. It stores the output into id, and then makes a new TreeNode, adding it to the line's list of children. If the extra character that was read was an A, then the node is an actor. If it was an L, then the node is a line. If it was neither, then it raises an error, since it cannot understand what the type is supposed to be. After assigning the type to the node, it then adds id to the dictionary, using the key 'id'.

```

private void ReadID(StreamReader line, TreeNode lineTree)
{
    char character = ReadChar(line); // Read the next character
    int id = ReadNumber(line); // Read the number attached to it
    TreeNode idNode = new TreeNode(); // Make a new ID node
    lineTree.AddChild(idNode); // Add it to the line's children

    if (character == 'A')
    { // If the ID is an actor
        idNode.AddType("actor");
    }
    else if (character == 'L')
    { // If the ID is a line
        idNode.AddType("line");
    }
    else
    {
        throw new Exception($"Tokenizer: Unable to decide if this is an actor or line reference: {character}");
    }
    idNode.value.Add("id", id.ToString()); // Add the number to the node's dictionary
}

```

Internally, `ReadNumber()` peeks the next character using `PeekChar()` and checks whether the read character is a number. `PeekChar()` functions similarly to `ReadChar()`, but instead of consuming the character, it leaves it in the stream. If the peeked character is a number, then it reads it properly and appends it to the number as a string. When it finally reaches a character that isn't a number, it will then attempt to parse the number into an integer, which it will promptly return.

```

private int ReadNumber(StreamReader line)
{
    string num = "";
    while (!line.EndOfStream)
    { // While the end of the line hasn't been reached
        char character = PeekChar(line); // Peek a character
        if (int.TryParse(character.ToString(), out _))
        { // If it is a number
            num += ReadChar(line); // Read it and add it to the number string
        }
        else
        {
            break;
        }
    }
    int inted; // Prepare a variable to store the int version of the string
    int.TryParse(num, out inted); // Cast num into an int
    return inted; // Return the number
}

```

I realised that this code could be optimised. Since `num` was already confirmed to contain only numbers, there was no need to try to parse. Instead, `int.Parse()` could be used instead, saving an extra 32-bit number, and reducing the game's memory footprint.

```
return int.Parse(num); // Return the number
```

The other dispatcher functions work similarly, reading off the extra characters they need, and making a new node that stores the important information, after which, this node is added to the line's child list. The only exception is ReadIf(), which must read a variable and an operator first before deciding what to do next. Both of these dispatched functions will make a node and add it as a child of ifNode. The next character is then checked whether it is a number or a letter; if it is a number, then ReadNumber() reads it and adds it as a child of ifNode. If it is a variable, ReadVariable() reads it off instead.

```
private void ReadIf(StreamReader line, TreeNode lineTree)
{
    TreeNode ifNode = new TreeNode(); // Make a new node
    lineTree.AddChild(ifNode); // Add it the lineTree's children
    ifNode.AddType("if"); // Set the type to 'if'
    ReadVariable(line, ifNode); // Read the variable and add it to ifNode's children
    ReadOperator(line, ifNode); // Read the operator and add it to ifNode's children
    char character = PeekChar(line); // Peek at the next character
    if (char.IsNumber(character))
    { // If it is a number
        TreeNode numNode = new TreeNode();
        ifNode.AddChild(numNode); // Make a new child node to store the number
        numNode.AddType("num"); // Set the type to 'num'
        numNode.value.Add("num", ReadNumber(line).ToString()); // Assign the number to numNode
    }
    else if (char.IsLetter(character) || character == '_')
    { // If the next character is a letter or _
        ReadVariable(line, ifNode); // Read a variable
    }
    else
    {
        throw new Exception("Tokenizer: Nothing valid to compare to.");
    }
}
```

After the line has been tokenized in its entirety, it will then pass the output to Analyzer(), which will read the contents and will do the actual interpretation. First, it will find out how many dialogue lines are needed to be fetched and also make a version for the return value of Analyzer(). It will then analyse each child of fileTree, which requires analysing each child element within the line.

```
private (string, string, string[]) Analyzer(TreeNode fileTree)
{
    string[] lines = new string[fileTree.GetChildren().Count]; // Make a new return array based on the number of lines
    string returnType = "none"; // Default return type
    string[] returnLines = new string[lines.Length]; // Copy of lines
    for (int i = 0; i < fileTree.GetChildren().Count; i++)
    { // For each line
        TreeNode line = fileTree.GetChildren()[i]; // Get the line
        List<TreeNode> elements = line.GetChildren(); // Get the elements of the line
        if (elements[0].GetNodeType() == "actor")
```

If the first node is an actor, the second node is a separation marker, and the third node is a line node, then returnType is 'line'. This means only 1 line should be returned and as such all the lines need to be analysed properly, by querying the database and converting the output in LineAnalysis(). LineAnalysis() will return only 1 string, since it

concatenates all the lines supplied to it. Just before it returns the line, it also removes the trailing space that will be there.

```
if (elements[1].GetNodeType() == "sep" && elements[2].GetNodeType() == "line")
{
    // If a separator and a line node follow the actor
    returnType = "line"; // Set the return type to line
    returnLines[i] = LineAnalysis(elements.GetRange(2, elements.Count - 2)); // Analyse all the lines into 1
    string returnActor = RequestNext(false, "actor", Int32.Parse(elements[0].value["id"])); // Get the actor from the database
    return (returnType, returnActor, returnLines); // Return the values
}
else
{
    foreach (TreeNode element in elements){
        Debug.Log(element.GetNodeType());
    }
    throw new Exception("Analysis: Unable to progress after discovering actor in analysis.");
}
private string LineAnalysis(List<TreeNode> lines)
{
    string returnLine = ""; // Store the output line
    for (int j = 0; j < lines.Count; j++)
    {
        // For every dialogue line
        returnLine += RequestNext(true, "english", Int32.Parse(lines[j].value["id"])) + " "; // Request it from the database then append it the output
    }
    returnLine = returnLine[..^1]; // Remove the trailing space
    return returnLine;
}
```

If the first node is instead a jump statement, returnType will be set to true to signal to the parent function, RequestNextLine(), that another line will need to be read. It will then use JumpToSection() to set lineNumber to the new line that the interpreter needs to start at.

```
else if (elements[0].GetNodeType() == "jump")
{
    returnType = "true"; // Tell RequestNextLine that it needs to jump to another line
    JumpToSection(elements[0]); // Get the line number of the section
    return (returnType, null, null);
}
```

JumpToSection() simply checks the dictionary, dialogueSections, for the correct line number, and proceeds to the next line.

```
private void JumpToSection(TreeNode jumpNode)
{
    string section = jumpNode.value["section"]; // Get the line number of the section
    lineNumber = dialogueSections[section] + 1; // Set the new line number
}
```

If the node is an if statement, then Analyzer() may need to signal that another line is needed. It first reads the third element, which is guaranteed to be a variable, and Analyself() will hand the correcting parsing of this. If the if node is instead signalling that it is assigning a variable, the variable is retrieved. This variable may be choice, which is the internal marker for the node, in which case, the value calculated at the start of Analyself() will be assigned to it. However, most likely, the variable is likely to be one in the save data, in which case, the game manager needs to use to request the assignment of the variable.

```

else if (elements[0].GetNodeType() == "if")
{
    returnType = "false";
    int var1 = AnalyseIf(elements[2]); // Analyse the third element
    if (elements[1].GetNodeType() == "assign")
    { // If the element is getting assigned
        if (elements[2].value["var"] == "choice")
        { // If the third element was the variable choice
            choice = var1; // Set choice to the value of var1
        }
        else
        {
            gameManager.SetVarValue(elements[2].value["var"], var1); // Tell the game manager to assign the variable in the save data
        }
    }
}

```

The game manager uses a special C# feature called reflection to work out which variable of the save data it is supposed to assign, based upon the name of the variable it is given. Reflection works by reading the metadata of the source code and attempts to find the relevant property from there. If the variable is found within the source code of WorldFlags, then saveData.worldFlags has the value set to the supplied variable.

```

public void SetVarValue(string variableName, int value)
{ // Sets the value of the variable named within itself
    Type type = typeof(WorldFlags); // Gets the id of the type of WorldFlags
    PropertyInfo? property = type.GetProperty(variableName); // Potentially null output from trying to find the variable
    if (property == null)
    {
        Debug.LogError($"{variableName} is not a variable of the manager.");
        // Do nothing to not cause havoc
    }
    else
    {
        property.SetValue(saveData, value); // save the changed value
    }
}

```

If the node is instead a comparison, then the other variable, which is the 4 one in the list, will be compared to the other in EvalComparison(), which is a switch statement based upon the operator supplied. The output of EvalComparison() will either be true or false, denoting whether the expression passed or failed. If it passed, then the jump node is read, and returnType is set to true, along with the new line number being fetched with JumpToSection().

```

else if (elements[1].GetNodeType() == "comp")
{ // If the variable is being compared
    int var2 = AnalyseIf(elements[3]); // Analyse the next variable
    bool passed = EvalComparison(var1, elements[1].value["op"], var2); // Compare the variables
    if (passed)
    { // If the comparison was true
        if (elements[4].GetNodeType() == "jump")
        { // If the next node is a jump node
            JumpToSection(elements[4]); // Set the line number to the section mentioned
            returnType = "true"; // Tell RequestNextLine that it needs to jump to another line
        }
        else
        {
            throw new Exception("Analysis: Expected a 'jump' node with a section.");
        }
    }
}

```

Within Analyse(), the function checks whether the node supplied is a variable or a number. If it is variable, and it is ‘choice’, then the interaction specific variable is set. However, if it is not, then the game manager is contacted to retrieve the value of the variable from the save data.

```
public int GetVarValue(string variableName)
{
    // Gets the value of the variable named within itself from the world flags
    Type type = typeof(WorldFlags); // Gets the id of the type of WorldFlags
    PropertyInfo? property = type.GetProperty(variableName); // Potentially null output from trying to find the variable
    if (property == null)
    {
        Debug.LogError($"{variableName} is not a variable of the manager.");
        return -1; // Error code
    }
    else
    {
        return Convert.ToInt32(property.GetValue(saveData.worldFlags, null)); // Cast to an integer
    }
}
```

If the node is a number instead, then it is parsed and returned as such. This finalises the interpreter, but to call all this code, the interaction object first needs to receive an interaction request, which is handled with PlayerGunInteraction. That script already checks if the player is attempting to interact with a weapon, and so an extra check is added to check if the weapon is interactable. All interactable objects that aren’t weapons will inherit from the Interactable class, which sets the GameObject that it is assigned to have the Interact tag, while also defining an abstract Interact method. DialogueInteraction implements this method by retrieving a new line. If the output of the interpreter is of the type ‘line’, then it sent to the dialogue manager.

```
public override void Interact()
{
    (string, string, string[]) output = RetrieveNextLine(); // Get the next line
    if (output.Item1 == "line")
    {
        // If it is a line, send it to the dialogue manager
        Debug.Log($"Sending line '{output.Item2 + ":" + output.Item3[0]}'");
        dialogueManager.DisplayLine(output.Item2 + ":" + output.Item3[0]);
    }
}
```

The dialogue manager contains a reference to the subtitle text object, and a list of the lines displayed on it. Each time a new line needs to be displayed, it checks whether the maximum lines has been reached, and if it has, it removes the first one and stops the fade-out timer that is attached to it, which automatically removes the line after a set amount of time.

```
public void DisplayLine(string text)
{
    Debug.Log($"Sent text: '{text}'");
    subtitleCanvas.GetComponent<CanvasGroup>().alpha = 1; // Show the subtitle box
    if (lines.Count >= maxLines)
    { // If too many lines are shown
        lines.Remove(lines[0]); // Remove the starting one
        StopCoroutine(Timer(lines[0]));
    }
    lines.Add(text); // Add the new line
    StartCoroutine(Timer(text)); // Start a fade out time for that line
}
private IEnumerator Timer(string text)
{
    yield return new WaitForSeconds(fadeAwayTimer); // Wait some seconds before the line is removed
    lines.Remove(text); // Remove the line
}
```

I consulted SillyAustralian on the time it should take for the line to fade away:

#### *Interview*

SillyAustralian: The text in the subtitle box should fade away after 2 seconds and should also remain visible in the pause menu. Also note that the deadline is almost up, so finish the testing for this feature and leave it at that.

Stakeholder Signature:



The final thing to touch up on was what the dialogue manager had to do every frame, which was assigning the contents of the subtitle window, or hiding it if no text was there. This is done with a for loop which pieces together the string that will be shown inside the UI element.

```

void Update()
{
    if (lines.Count == 0)
    { // If no lines are shown
        subtitleCanvas.GetComponent<CanvasGroup>().alpha = 0; // Hide the canvas
        return;
    }
    string text = "";
    foreach (string line in lines)
    { // For every line
        text += line + "\n"; // Construct the line that goes in the text object
    }
    textObj.text = text; // Assign the text inside
}

```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Requesting a line	The interaction will always attempt read a normal line, and so this must be tested correctly	The line is put onto the subtitle window, which is displayed, and then fades away after 2 seconds	Passed	
Valid	Reading a line made of multiple pieces	Certain pieces of text may need to be repeated, and this can be made easier by requesting multiple lines from the database	The lines are pieced together and put onto the subtitle window, which is displayed, and then fades away after 2 seconds	Passed	
Boundary	Reading a second line after reading 1	The interaction will rarely end with one click, and this needs to be handled properly	The second line is added below the first in the subtitle menu, and fades away after 2	Failed: the interpreter skipped a line	Caused by skipping the lines incorrectly

			seconds		
Error	Requesting a line or actor that doesn't exist in the database	The developer may forget to fill in the database appropriately, and this should be handled	An error is raised	Passed	
Error	Reading a line that is lacking information	The developer may forget to complete the .dlg file, this must not be passed to the rest of the tokenizer	An error is raised	Passed	
Error	Reading an unrecognised token	If the developer has not written the .dlg file properly, the tokenizer should warn them	An error is raised, specifying information on the error	Passed	

When reading multiple lines, the tokenizer is skipping lines. This can be solved by closing the file each time the tokenizer finishes, since the problem is caused by the StreamReader still being in memory when the tokenizer is run again. The best way to do this is the using statement, which automatically closes the file when outside it.

```
TreeNode fileTree = new ...TreeNode(); // This is here in case a choice exists,
using (StreamReader fileReader = new StreamReader(filepath)){
```

Type	What data fits criteria	Justification	Expected	Actual	Comments
Boundary	Reading a second line after reading 1	The interaction will rarely end with one click, and this needs to be handled properly	The second line is added below the first in the subtitle menu, and fades away after 2 seconds	Passed	

## End of prototype tests

### *Transpiling a line*

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Reading normal syntax	The majority of the transpiler's job is to deal with the syntax, and this makes sense as a valid test	The syntax will be replaced with the appropriate value	Passed	
Valid	Reading a backslash	The transpiler must also be able to deal with an escaped character	The following character is skipped over	Passed	
Boundary	Reading a character that is not part of the syntax	The majority of the line will be like this, but the transpiler should not interpret it as syntax	The character is passed through the system	Passed	

### *Dialogue Interaction*

Type	What data fits criteria	Justification	Expected	Actual	Comments
Valid	Requesting a line	The interaction will always attempt read a normal line, and so this must be tested correctly	The line is put onto the subtitle window, which is displayed, and then fades away after 2 seconds	Passed	
Valid	Reading a line made of	Certain pieces of text	The lines are pieced	Passed	

	multiple pieces	may need to be repeated, and this can be made easier by requesting multiple lines from the database	together and put onto the subtitle window, which is displayed, and then fades away after 2 seconds		
Boundary	Reading a second line after reading 1	The interaction will rarely end with one click, and this needs to be handled properly	The second line is added below the first in the subtitle menu, and fades away after 2 seconds	Passed	
Error	Requesting a line or actor that doesn't exist in the database	The developer may forget to fill in the database appropriately, and this should be handled	An error is raised	Passed	
Error	Reading a line that is lacking information	The developer may forget to complete the .dlg file, this must not be passed to the rest of the tokenizer	An error is raised	Passed	
Error	Reading an unrecognised token	If the developer has not written the .dlg file properly, the tokenizer should warn them	An error is raised, specifying information on the error	Passed	

## Interview

**SillyAustralian:** Reading a line with an actor works well, including the optional extra of reading the multiple lines from the database in one line. Nothing else needs to be added.

Stakeholder Signature:



## Evaluation

### Evaluative Testing

Since the game is in a playable state, these tests will all be completed by both stakeholders and also filled in by them, unless specified otherwise.

### Robustness

These tests will make sure that the game fulfils the minimum hardware requirements, and this will be possible to test since SillyAustralian runs a game development company. These companies have access to many machines that fulfil different expected hardware combinations and some of these that follow the typical hardware detailed in the success criteria will be used to run these tests. Making sure that the game never dips below 30 fps will be achieved using the logging option in the FPS counter, which can be analysed afterwards to make sure the game doesn't dip far below 30.

Type	What data fits criteria	Justification	Expected	Actual
Normal	Play the game in the normal scene, with few elements	The player will typically experience the game like this, with a reasonable number of objects	The game not run below 30 fps	Pass – The game typically ran at 1600 fps
Intense	Playing the	In case the player is	The game not	Pass – The

	game in a scene with lots of entities	chased by many enemies, the game should be able to handle this well	run below 30 fps	game typically ran at 1250 fps
--	---------------------------------------	---	------------------	--------------------------------

*Evidence – Normal*

```
C:\> Users > eshan > AppData > LocalLow > RandomEF > Abrupt Animus > Player.log
51685 883.8616
51686 1126.76
51687 394.493
51688 579.2057
51689 848.6838
51690 1361.65
51691 1443.21
51692 1169.868
51693 1433.695
51694 1355.19
51695 1354.826
51696 1465.634
51697 1237.938
51698 1290.323
51699 1398.465
51700 1281.22
51701 1563.726
51702 1265.821
51703 785.1778
51704 1125.494
51705 1280.486
51706 1494.33
51707 1109.26
51708 1317.18
51709 1233.038
51710 1417.636
51711 683.996
51712 1153.403
51713 1612.376
51714 1356.115
51715 1428.991
```

*Evidence – Intense*

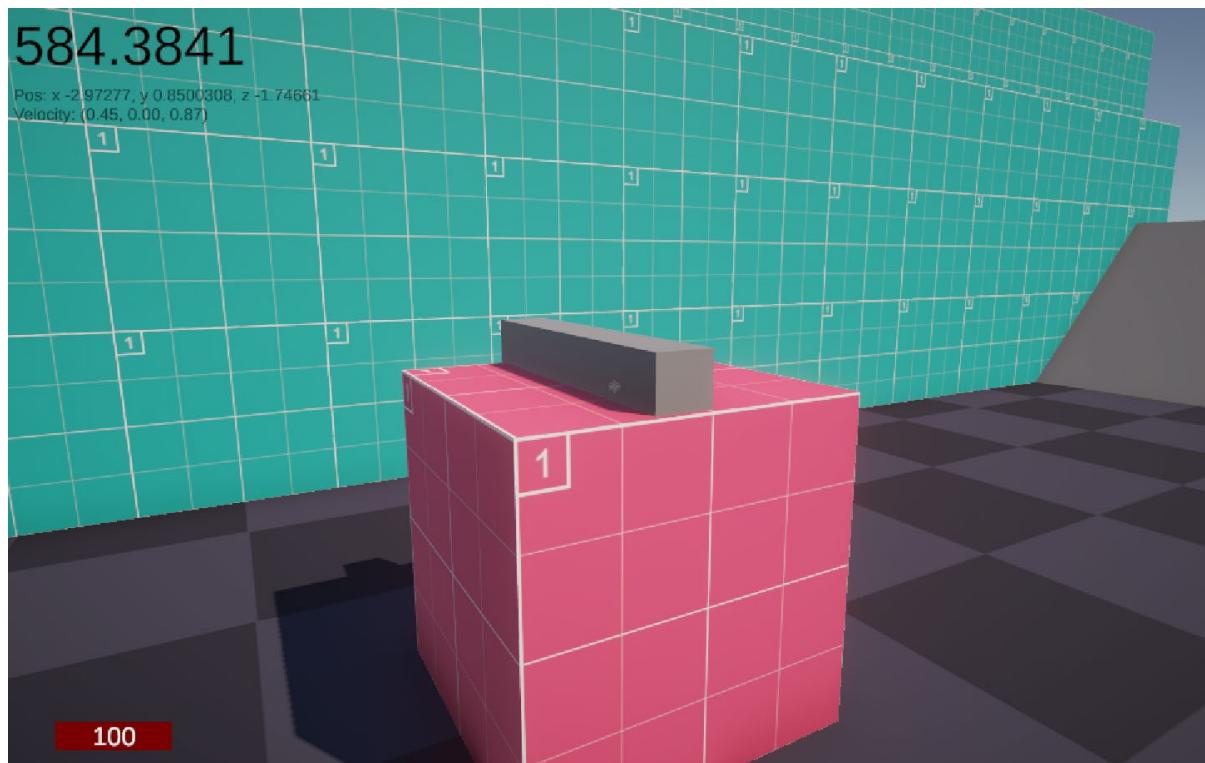
```
C:\> Users > eshan > AppData > LocalLow > RandomEF > Abrupt Animus > Player.log
55598 744.3823
55599 698.2233
55600 1075.159
55601 813.3389
55602 1183.148
55603 840.4803
55604 850.2639
55605 1123.979
55606 1166.175
55607 1082.375
55608 1251.408
55609 1570.6
55610 1044.165
55611 1346.077
55612 1296.677
55613 1282.221
55614 1187.649
55615 854.7702
55616 1194.889
55617 1310.793
55618 1340.833
55619 1423.284
55620 1144.563
55621 1090.033
55622 1063.491
55623 1059.658
55624 987.2675
55625 1501.948
55626 1152.743
55627 1089.797
55628 1068.031
55629 887.8654
55630 1178.406
55631 1033.914
55632 869.1849
55633 988.3391
55634 997.7094
55635 1292.651
55636 1270.813
55637 799.7416
55638 1072.846
55639 1269.841
55640 1067.578
55641 1265.344
55642 1314.75
55643 1092.983
55644 1429.179
```

## Glitches

The tests make sure that the player does not have a frustrating experience and will all be completed by the CEO of DrunkDriving, Amarveer Flora. These will be marked with the severity, from mild, to medium, to severe, and finally game-breaking.

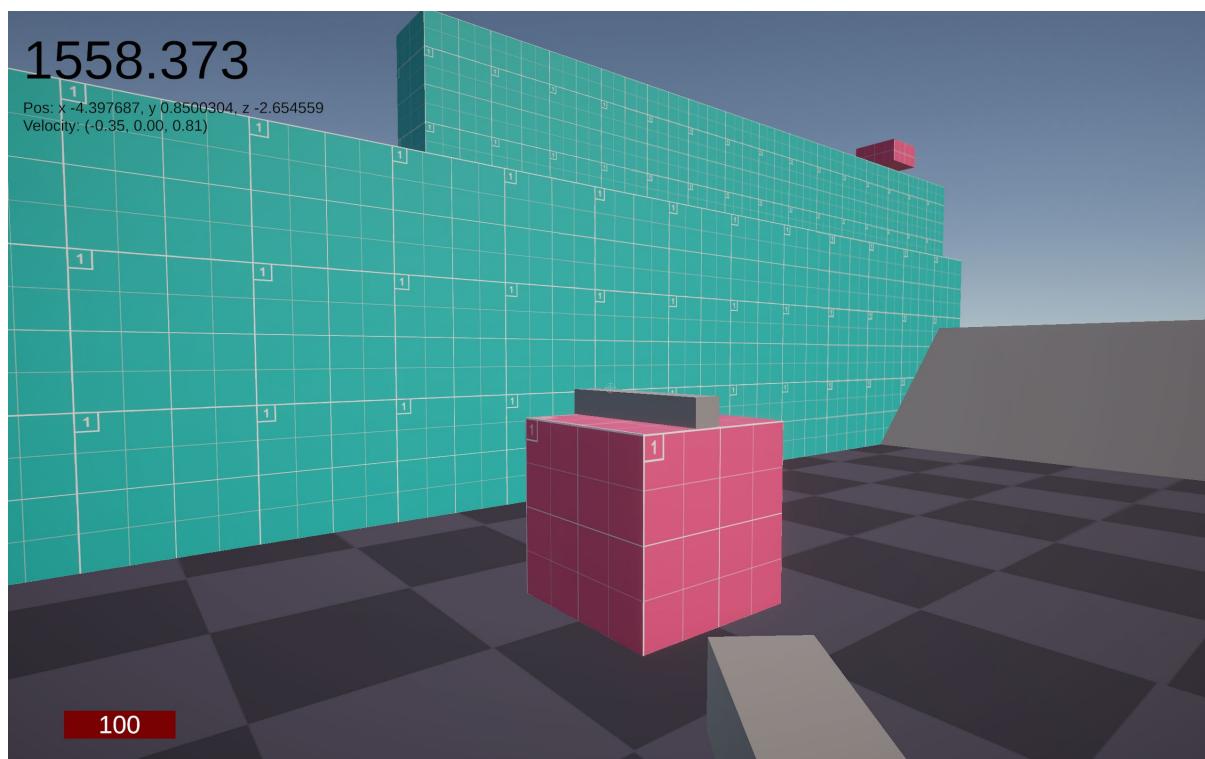
Type	Justification	Example	Severity	Steps to reproduce
Input	Any glitches with the player's input will be unsatisfactory, and may frustrate the player	The movement is jittery when the playing is not holding any input	Mild	Press any strafing input and then don't press another
Loading	Any glitches that affect the loading of the game can cause unpleasant results	When loading a save, any weapons that the player picked up in the world respawn	Mild	Pick up a weapon in the current levels, save, and then load again
UI	Glitches that affect the UI can range from being an inconvenience to game breaking	The HUD disappears when the player uses load within the pause menu; it can be fixed by opening and closing the pause menu	Medium	Open the pause menu, press load
Dialogue	Glitches that deal with the dialogue system can be very difficult to debug, but do not tend to break the game	The system sometimes reads the dialogue section as a line and fails	Mild	Have an if statement that fails with a dialogue section header right after

### *Jittery movement*



*(The player still has a small amount of velocity despite not holding any input for the past couple of seconds, see in the top left)*

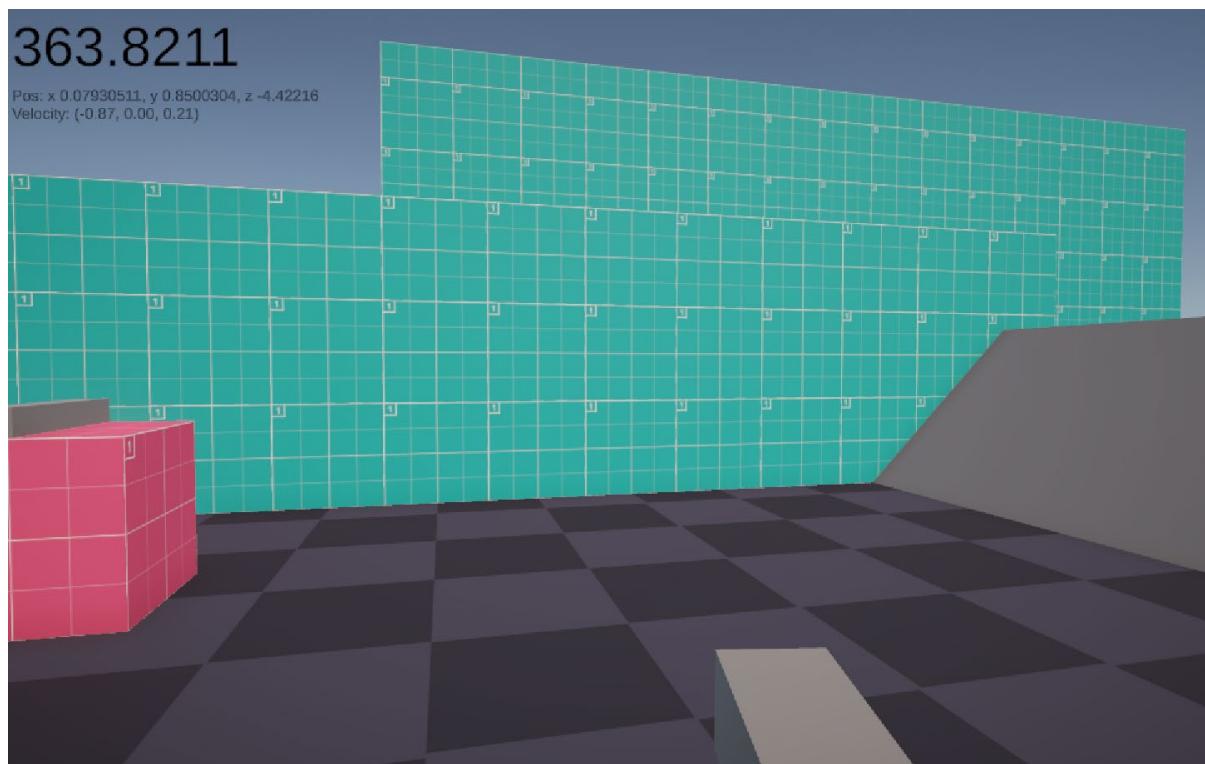
This bug is somewhat annoying for the player's experience but only affects them when there are no inputs applied, which is why this has been marked as being of mild severity. This issue exists with one of the functions related to movement, such as FrictionMultiplier() and its use in Movement(), which can be fixed in the future. This will affect the player's experience and could potentially change many properties with how the player moves, therefore meaning it should be one of the first bugs to be fixed.

*Weapon duplication*

(The player has already picked up the weapon in the centre, as it is the one the player is currently holding)

Caused by the loading progression and the way the save system works, this glitch is classified as mild severity, since it does not hinder the player's experience. In fact, it does the opposite – it gives the player more weapons to use. This is unintended, however, and has hence been classified as a mild bug by Amarveer Flora. To solve this, every weapon in the scene needs to be given a unique ID and be given to the player's weapon slots instead of the weapons being instantiated. This would require a minor rework of the save system and the contents of `ApplyData()`, which is where the weapons are given to the player.

### *HUD Disappearance*



*(The HUD has disappeared)*

This one has been marked as medium severity due to it hindering the player's ability to see important information in the game. It seems to be caused by the action of reloading the scene, which leads to the HUD not being re-enabled. It was originally disabled when the pause menu was selected, which was itself disabled by the reloading of the level. MenuManager may be changing the currently opened menu before the scene is reloaded, which would close any open menu automatically. A scene reload does also not incur the SceneManager.sceneLoaded event, which would tell the game manager to open the correct menu. Moving forward, fixing this would require some changes to the activation order of the scripts. Interestingly, the HUD can be returned by opening the pause menu and then closing it, which brings it back due to the HUD being the last opened menu in the records of MenuManager. Due to this, the issue was only marked as medium severity and not severe, since the player is able to reopen the menu if they open the pause menu.

### *Dialogue if skip*

When the dialogue script has an if statement whose outcome results in false, the interpreter will typically move forwards in the script, resulting in the line number being incremented. However, when this new line is a dialogue section header, the interpreter will fail due to the characters used to signify a section header is not recognised by the tokenizer, and hence the following parts of the interpreter. This is a mild issue, since it

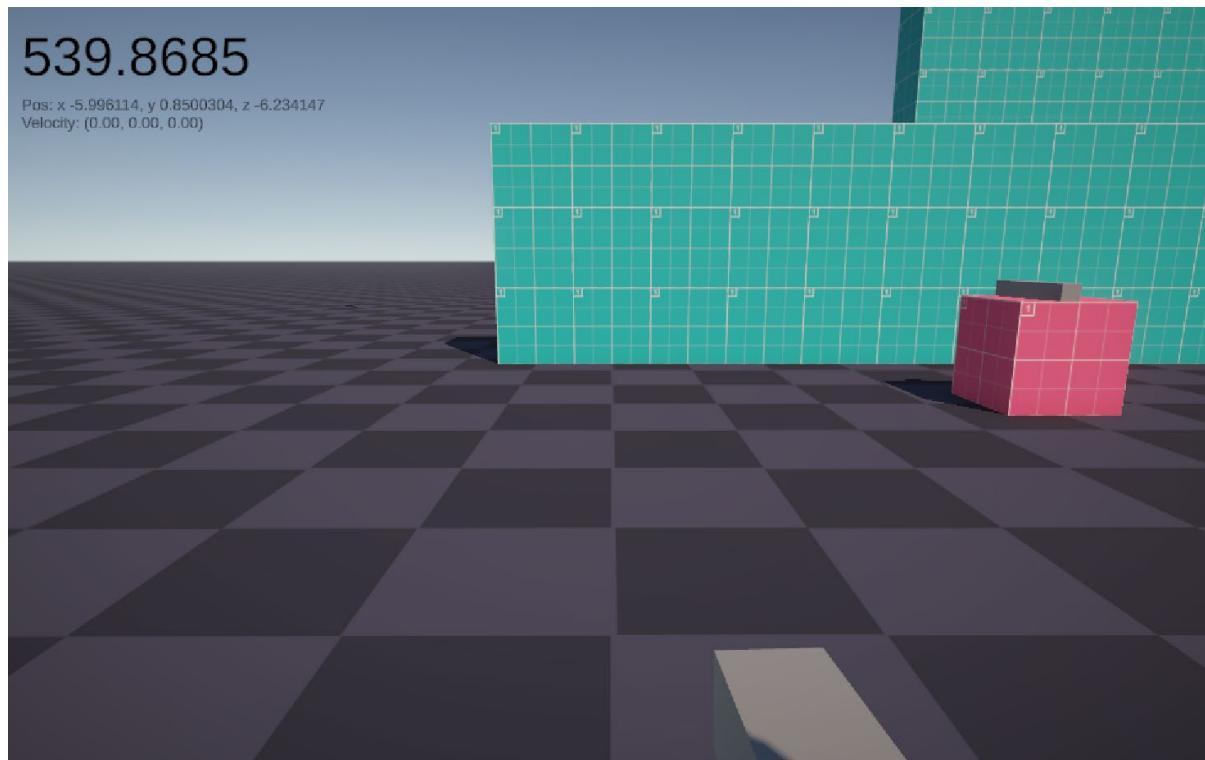
requires the interpreter to recognise the symbol and either move back a line or forward a line depending on where it is in the script. This can be added in the future by SillyAustralian's team and should not be that difficult of a fix. This has led to the bug being marked as a mild bug, especially since it only activates when the player reaches that point and does not hinder the player's ability to play the game normally, nor does it crash it.

## Strafing

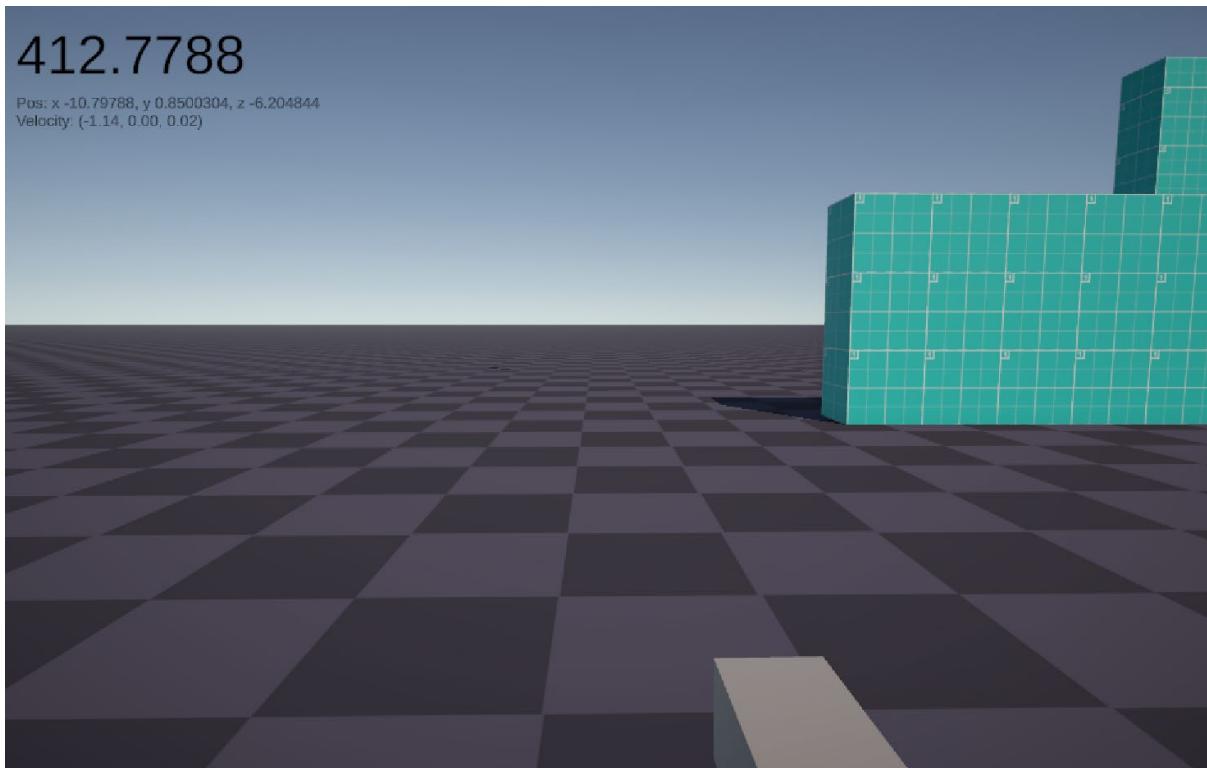
Tests for strafing will be evaluated using the floor, which will have a prototype texture across it. This prototype texture will accurately mark 1 metre intervals upon it. It can also be secondarily verified using the position shown in the debug screen in the top left corner. The HUD was disabled for these tests.

What data fits criteria	Justification	Expected	Pass/Partial/Fail	Comments
Pressing the left input, then releasing	These 8 tests relate to the different valid input directions that the player can supply, and every single one should work	The player moves left and stops	Pass	
Pressing the right input, then releasing		The player moves right and stops	Pass	
Pressing the up input, then releasing		The player moves up and stops	Pass	
Pressing the down input, then releasing		The player moves down and stops	Pass	
Pressing the left and up input, then releasing		The player moves north-west and stops	Pass	
Pressing the left and down input, then releasing		The player moves south-west and stops	Pass	
Pressing the right and up input, then releasing		The player moves north-east and stops	Pass	
Pressing the right and down input, then releasing		The player moves south-east and stops	Pass	
Pressing the	While not leading	The player	Pass	

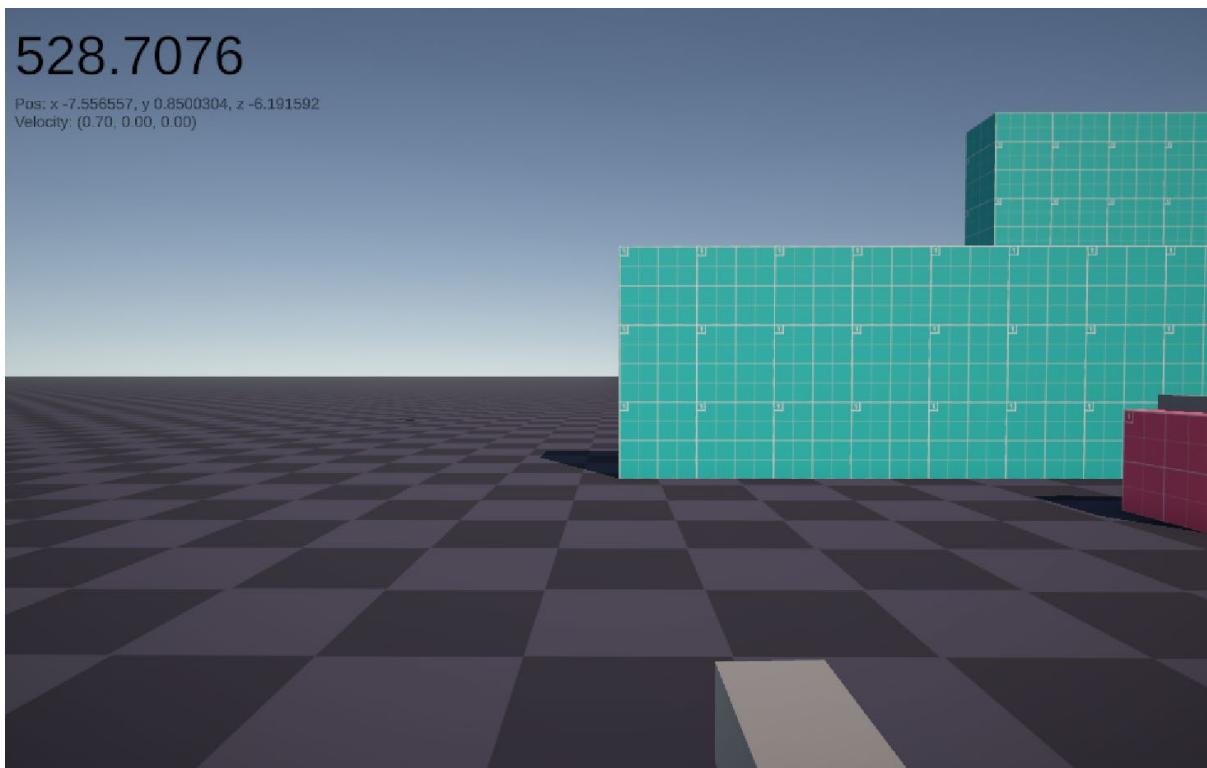
right and left input, then releasing	to any overall movement, it is an input the player can make that is made of completely valid inputs	doesn't move		
Pressing the up and down input, then releasing	The player doesn't move	Pass		
Pressing no input	Testing for the lack of inputs is also something that can happen on any frame, and must be tested correctly	The player doesn't move	Pass	
Pressing an unrelated key (e.g. M)	This is invalid and should not affect the strafing system, and must be dealt properly	The player doesn't move	Pass	



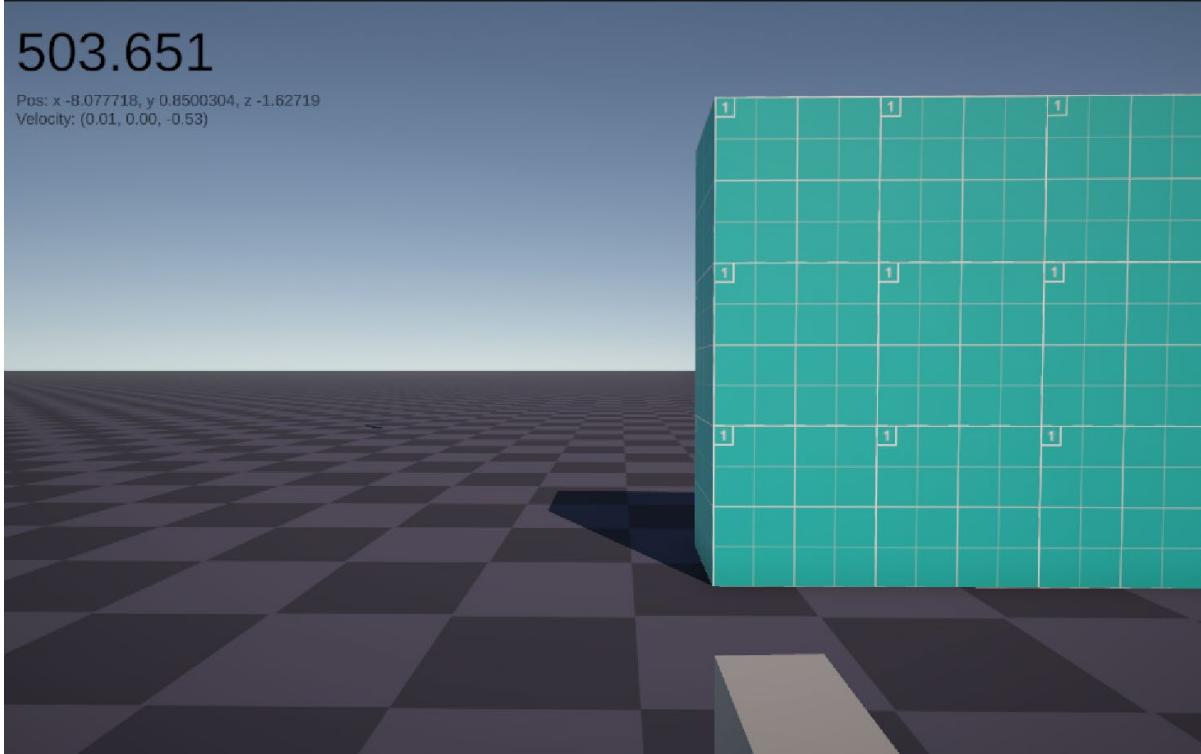
(The player starts here)



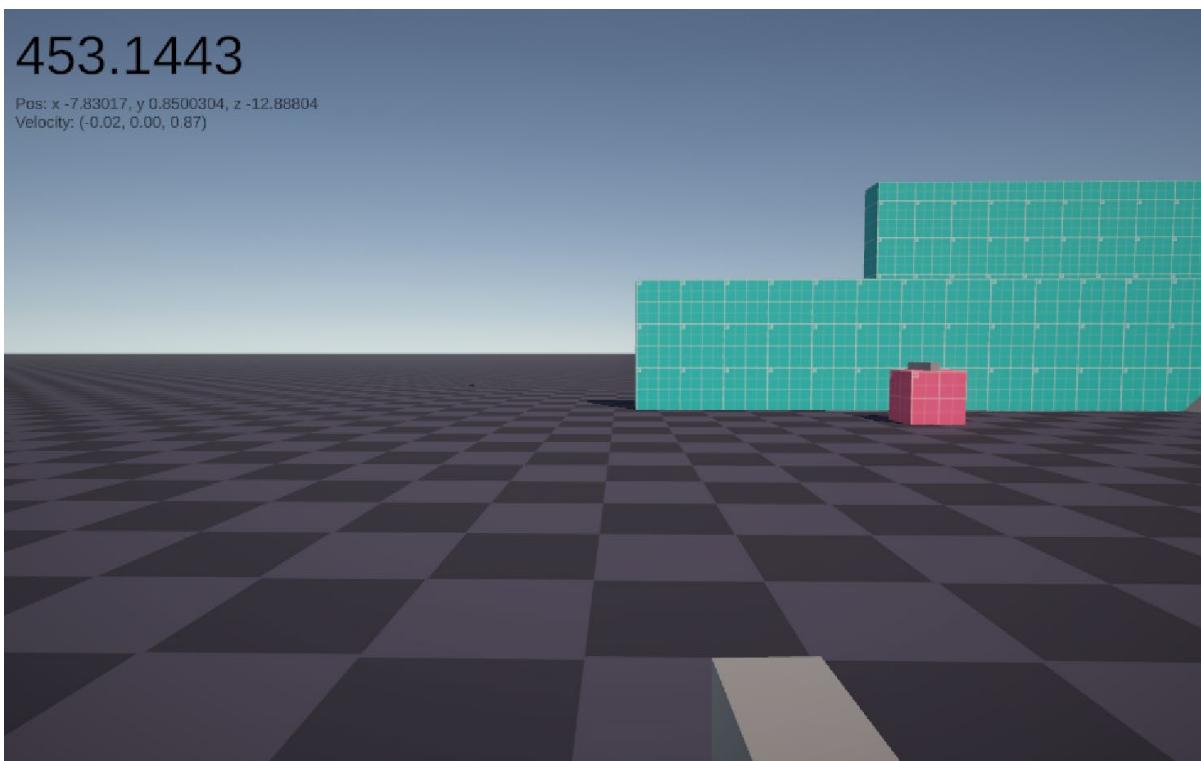
(Moving left)



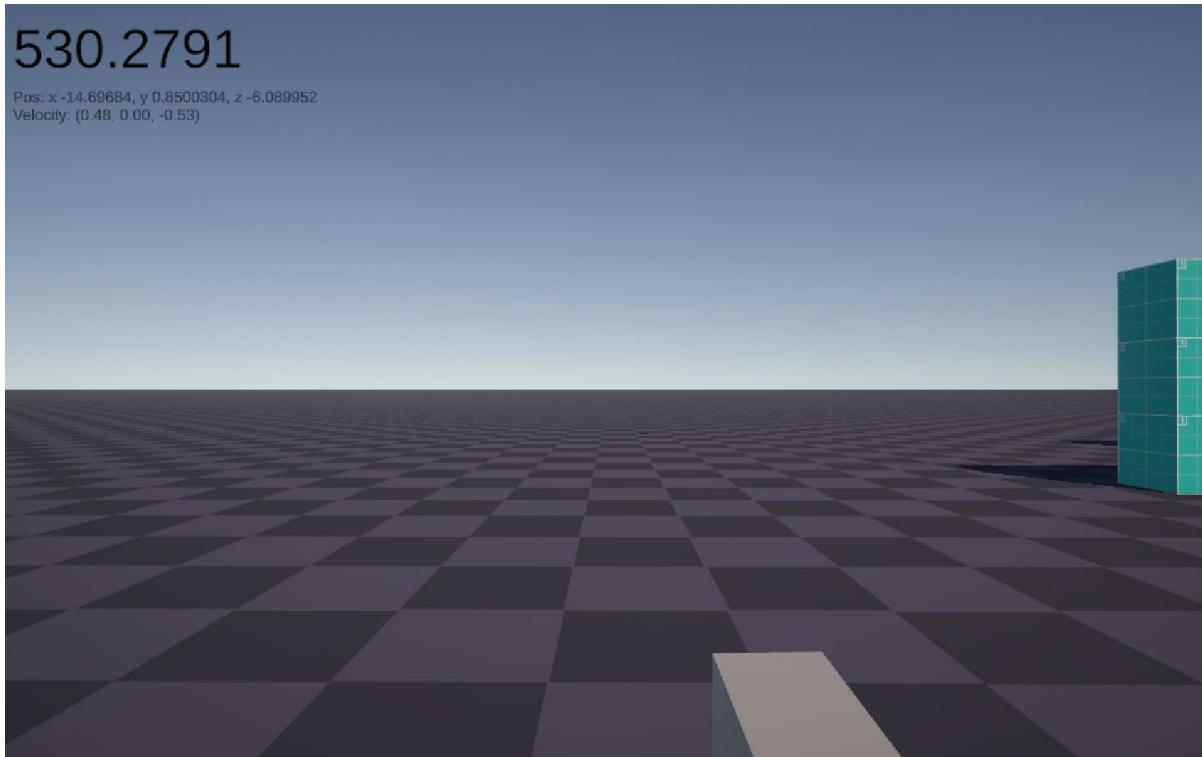
(Moving right)



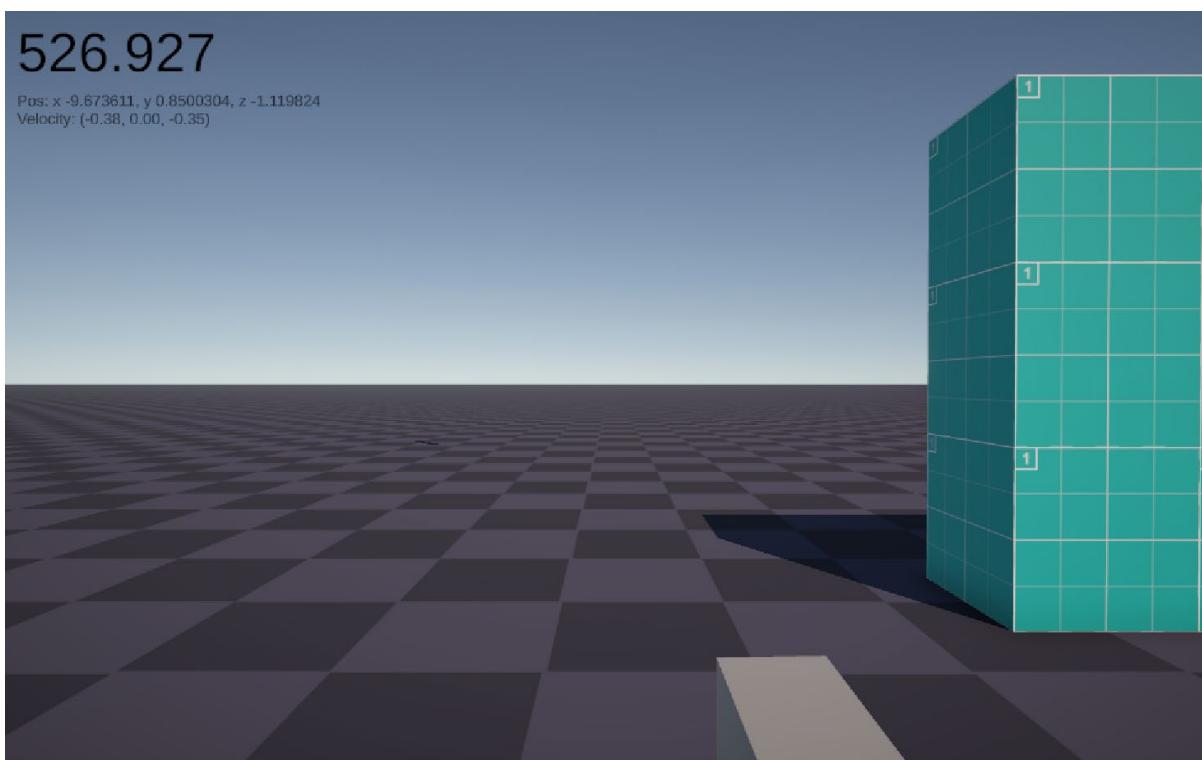
(Moving forwards)



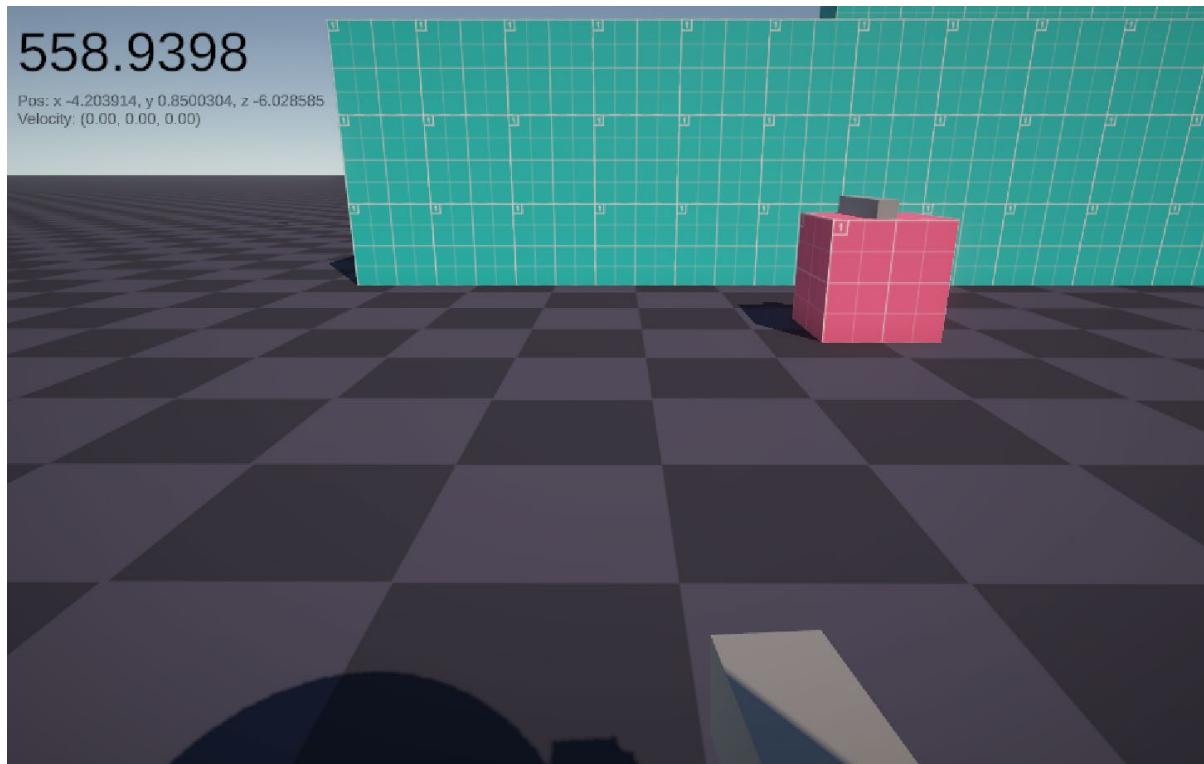
(Moving backwards)



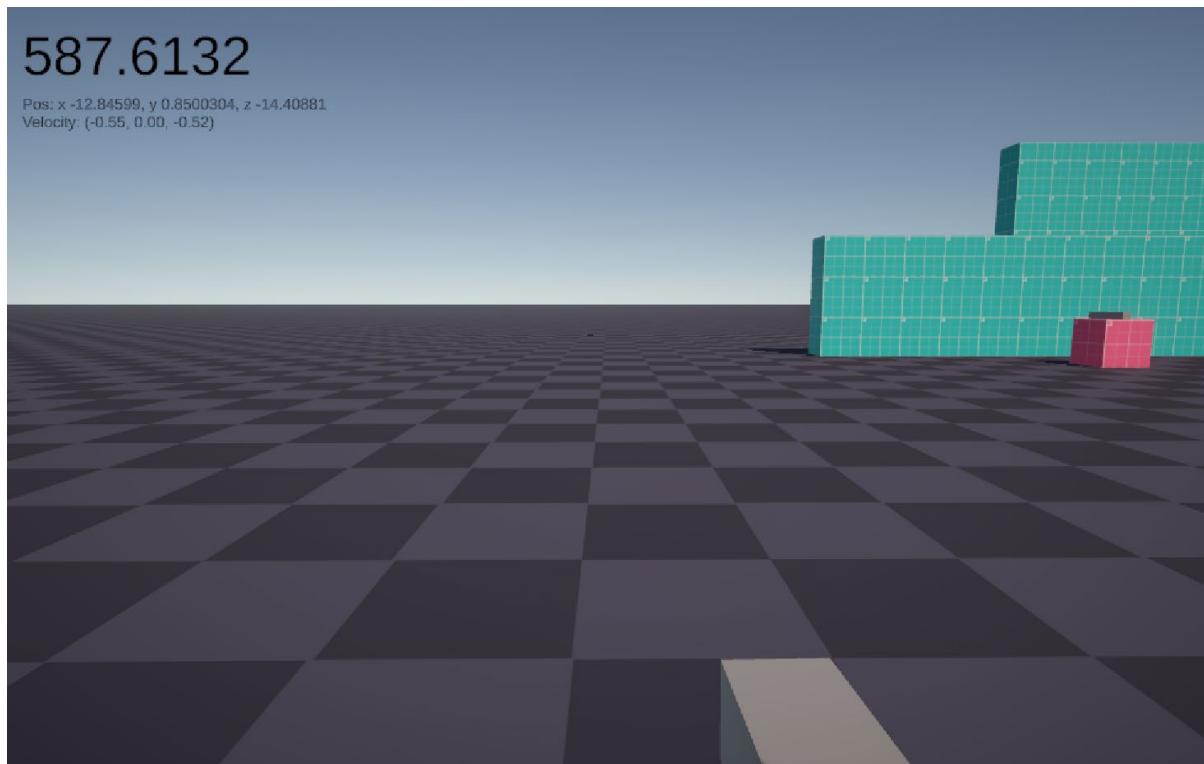
(Moving north-west)



(Moving north-east)



(Moving south-east)



(Moving south-west)

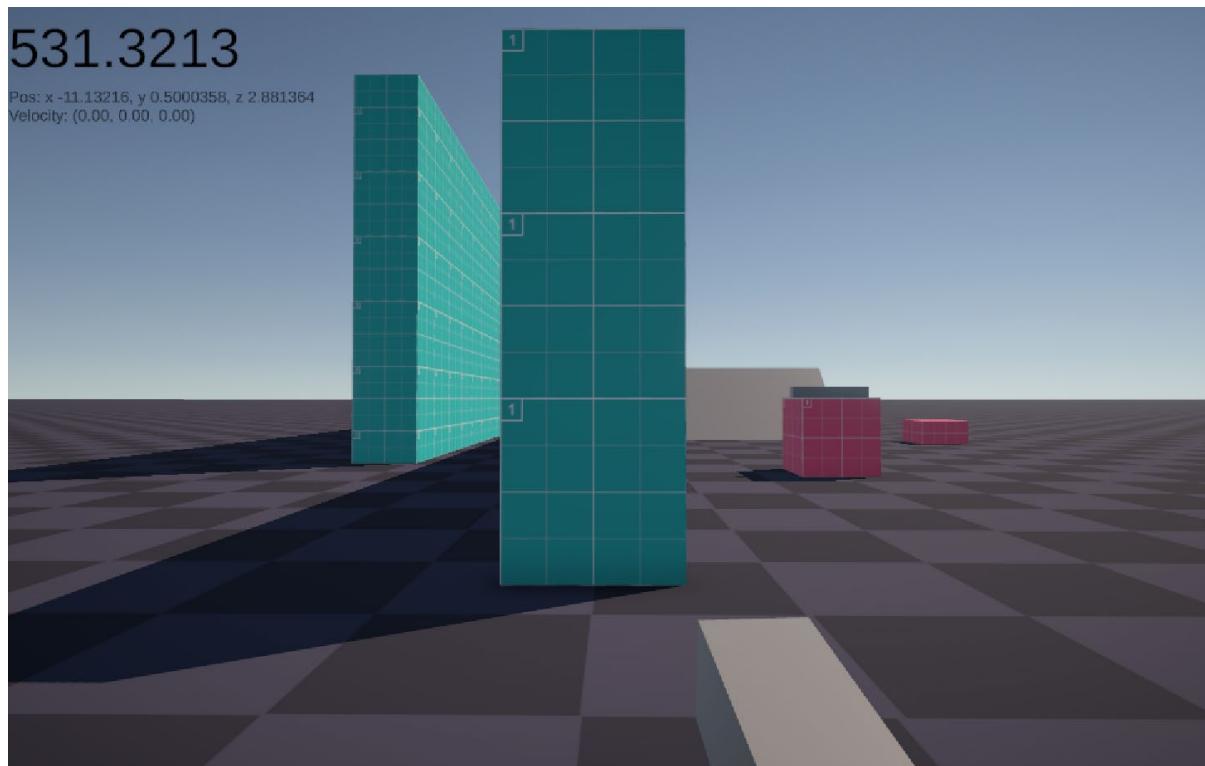
### Jumping

Tests for jumping will be evaluated using a wall, which will have a prototype texture across it. This prototype texture will accurately mark 1 metre intervals upon it. It can

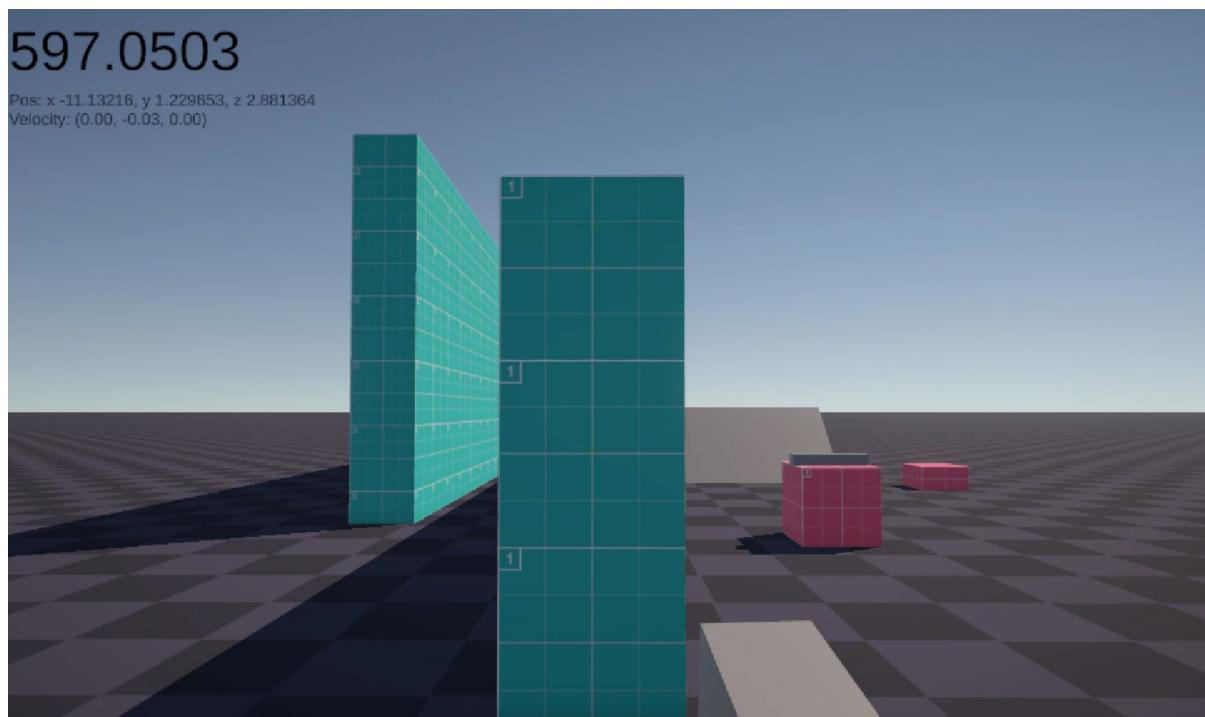
also be secondarily verified using the position shown in the debug screen in the top left corner. The HUD was disabled for these tests.

What data fits criteria	Justification	Expected	Pass/Partial/Fail	Comments
Pressing the jump input	This is the key the player will press when they want to jump up, and so should be tested correctly	The player moves upwards	Pass	
Pressing the jump input while in midair and moving upwards with air jumps remaining	This will test the player's ability to double jump, which will allow the player to reach slightly further distances	The player moves upwards	Pass	
Pressing the jump input while in midair and moving downwards		The player moves upwards	Pass	
Pressing the jump input while in midair and moving upwards with no air jumps remaining		The player continues moving upwards that they are moving – the input is effectively ignored	Pass	
Pressing the jump input while in midair and moving downwards with no air jumps remaining		The player continues moving downwards that they are moving – the input is effectively ignored	Pass	
Pressing no input	Testing for the lack of inputs is also something that can happen on any frame, and must be tested correctly	The player doesn't move	Pass	
Pressing an	This is invalid	The player	Pass	

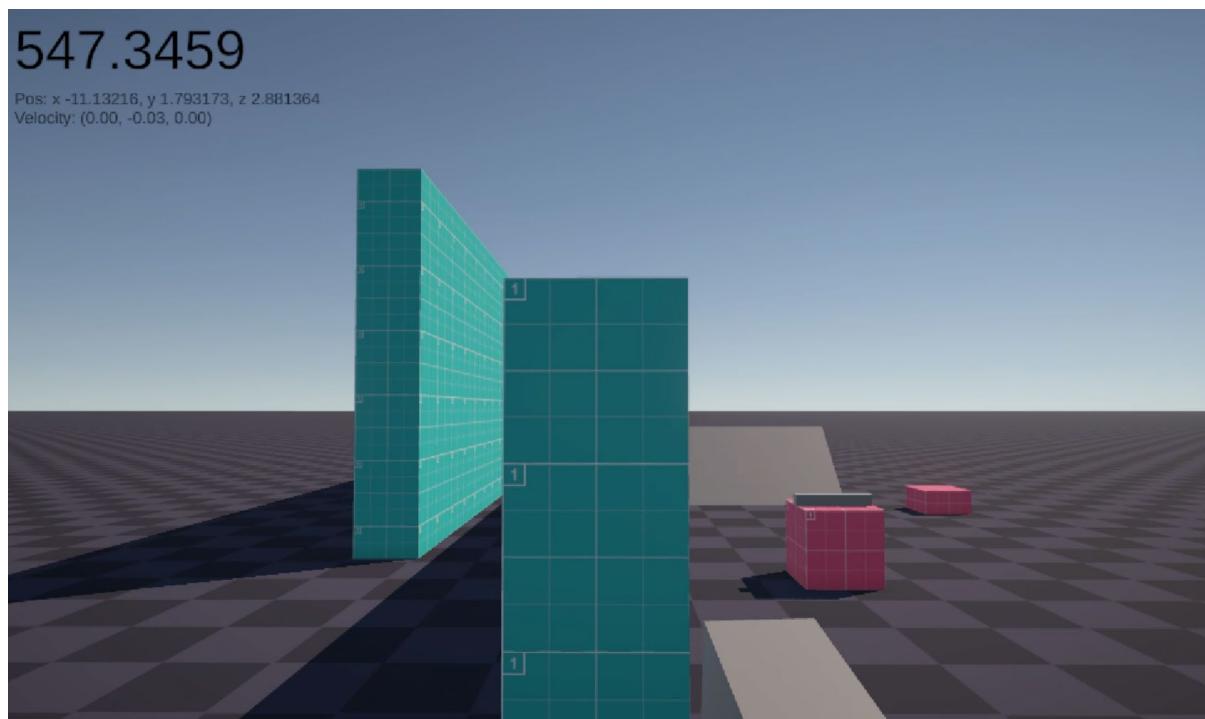
unrelated key (e.g. M)	and should not affect the jumping system, and must be dealt properly	doesn't move		
---------------------------	--	--------------	--	--



*(The player starts on the ground)*



*(The player jumps)*



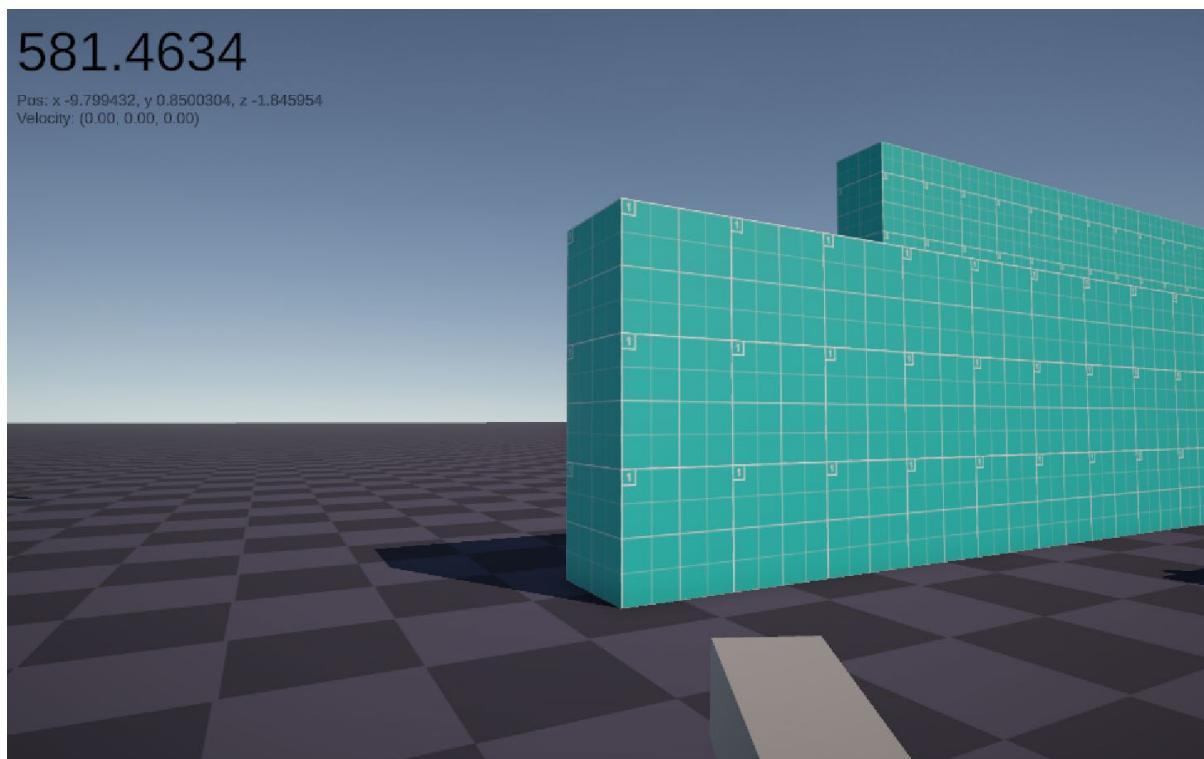
*(The player double jumps)*

## Crouching

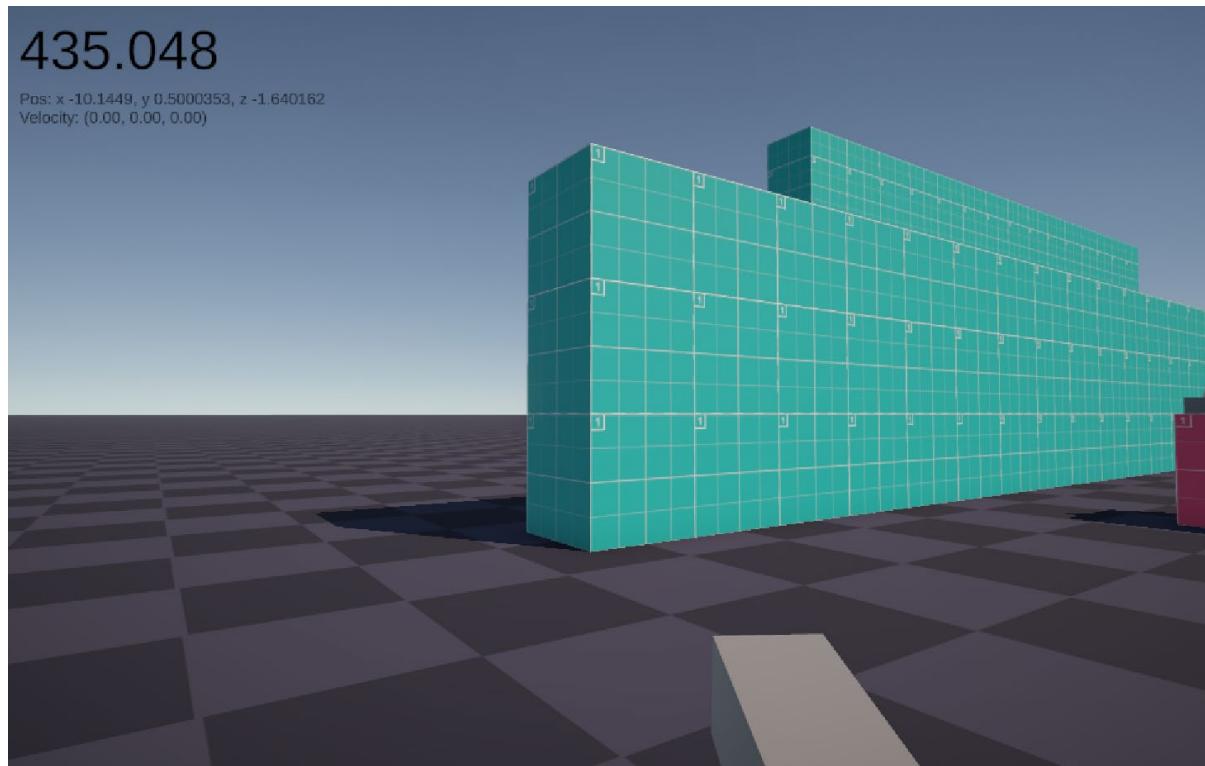
Tests for crouching will be evaluated using the same wall as the jumping tests. It can also be secondarily verified using the position shown in the debug screen in the top left corner. The HUD was disabled for these tests.

What data fits criteria	Justification	Expected	Pass/Partial/Fail	Comments
Pressing the toggle crouch, then releasing	This is one of the keys that the player can press to crouch	The player will shrink in height	Pass	
Pressing the hold crouch, then releasing	This is other key that the player can press to crouch, and is likely to be the more commonly used one	The player will shrink in height, and then return to the standing height	Pass	
Pressing the toggle crouch, then releasing. Then pressing the hold crouch, and	The player may use both inputs to crouch, and this should not lead to any conflicts	The player will shrink in height, and then return to the standing height when the hold crouch is	Pass	

then releasing it		released		
Pressing no input	Testing for the lack of inputs is also something that can happen on any frame, and must be tested correctly	The player remains at the standing height	Pass	
Pressing an unrelated key (e.g. M)	This is invalid and should not affect the crouching system, and must be dealt properly	The player remains at the standing height	Pass	



(The player starts tall)



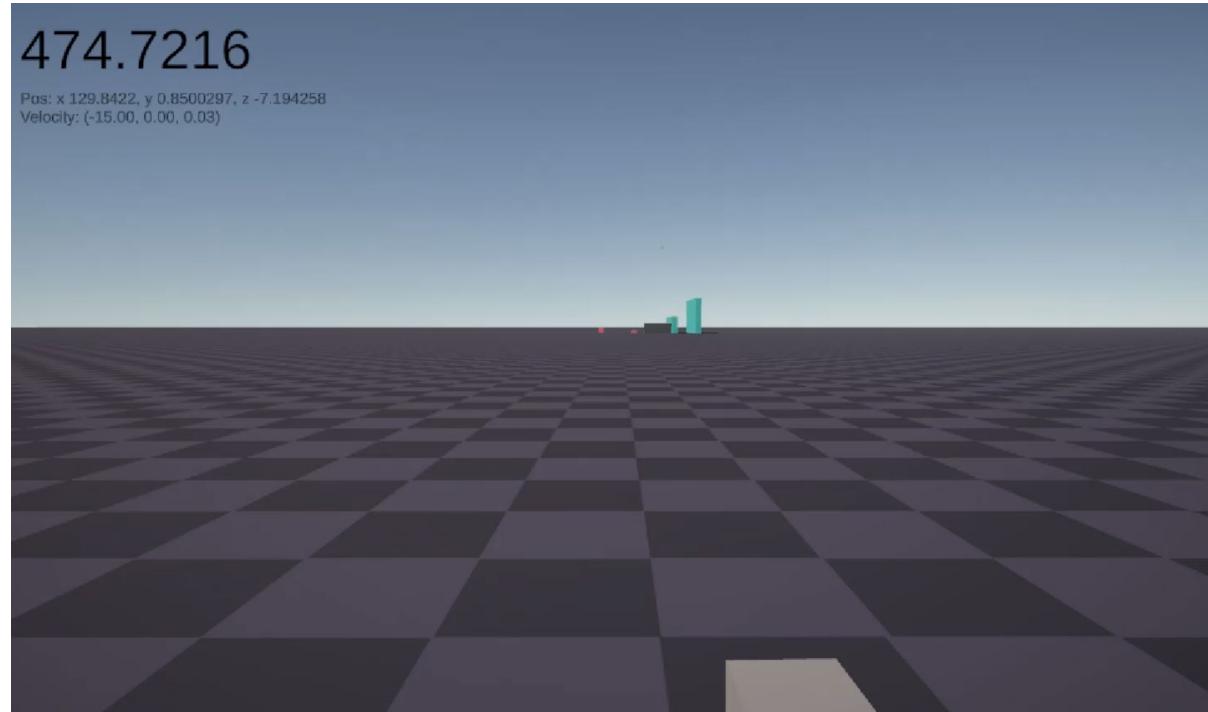
(The player crouches)

## Boosting

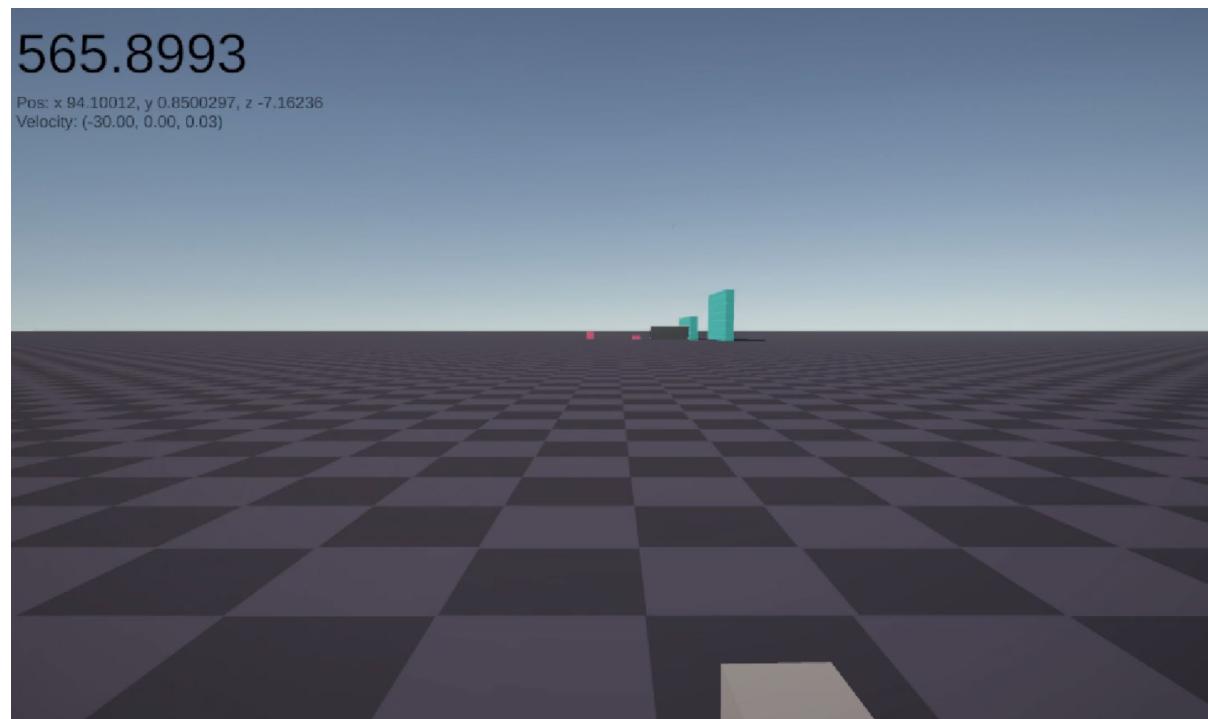
These will also be tested the same way as the strafing tests. The HUD was disabled for these tests.

What data fits criteria	Justification	Expected	Pass/Partial/Fail	Comments
Pressing the boost key while moving	This is the keys that the player can press to activate boosting	The player will double their current speed.	Pass	
Pressing the boost key while not moving	This will test how the system evaluates the player's speed	The player will not move	Pass	
Pressing no input	Testing for the lack of inputs is also something that can happen on any frame, and must be tested correctly	The player remains at the standing height	Pass	
Pressing an unrelated key (e.g. M)	This is invalid and should not affect the crouching	The player remains at the standing height	Pass	

	system, and must be dealt properly			
--	------------------------------------	--	--	--



(The player's maximum sprinting speed of 15 m/s)



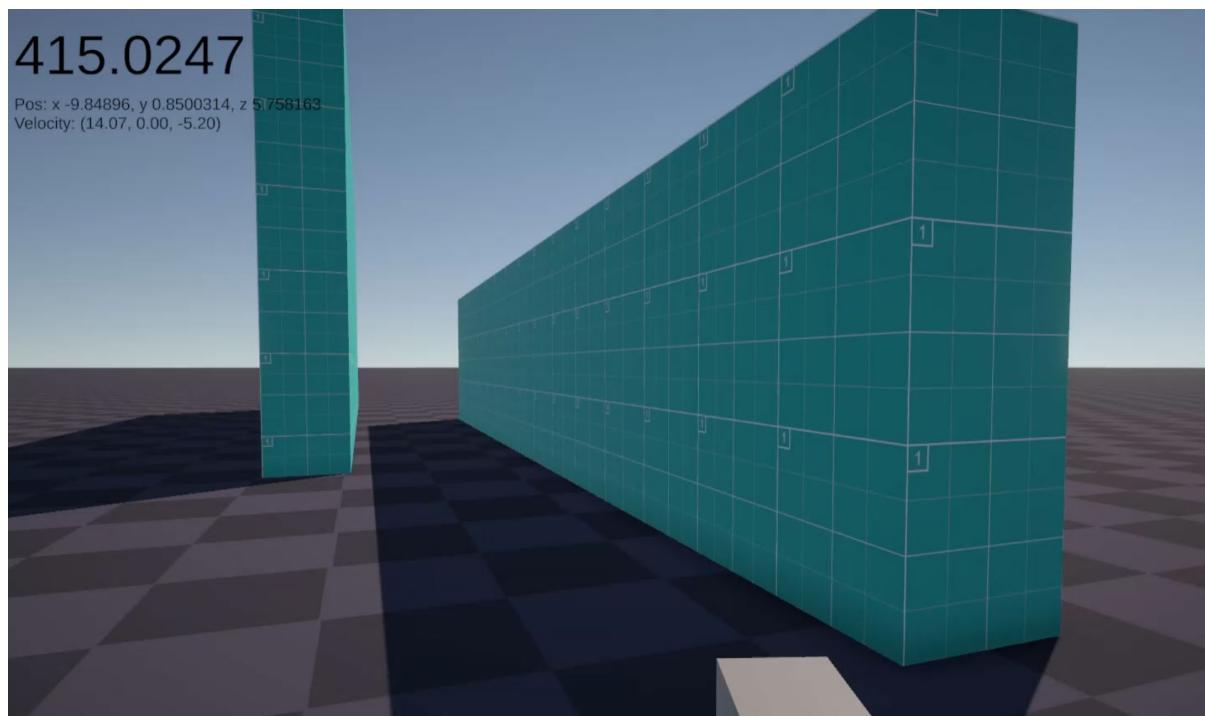
(The player's speed has doubled after pressing the boost button once)

## Wallrunning

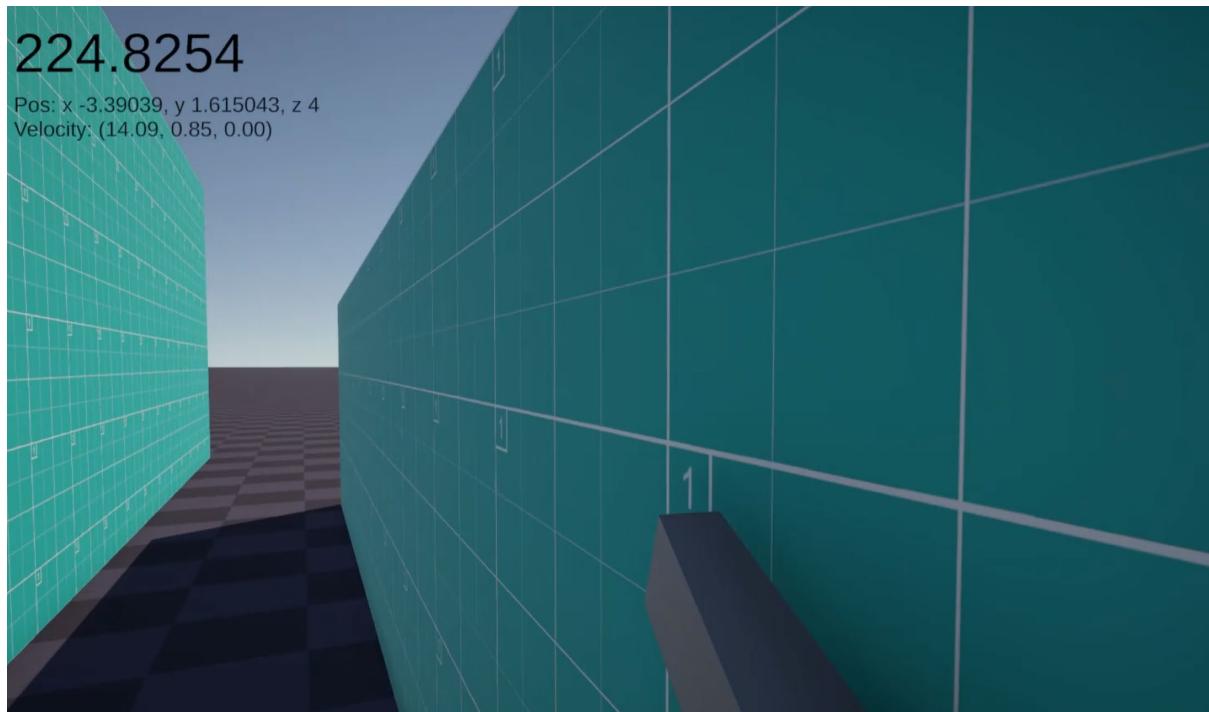
These tests are evaluated using 2 walls, of which the player will jump on and between them. The HUD was disabled for these tests.

What data fits criteria	Justification	Expected	Pass/Partial/Fail	Comments
Pressing forward on the wall	This is the most likely action that the player is going to do when they are on the wall, and as such, this should be the highest priority test	The player will move forward on the wall	Pass	It is not clear if the player has attached to the wall
Facing perpendicular to the wall	The player may want to dismount from the wall	The player will gain a speed boost as they disconnect from the wall and enter a walking state.	Pass	
The player pressing the jump key while on the wall	Since the aim of the game is to maintain speed, the player will want to jump away from the wall instead of the other methods to maximise the speed gain	The player will dismount from the wall and gain some speed in the direction of the wall normal and upwards	Pass	No feedback responding if an air jump has been used or the wall jump was used, other than verifying that the air jump existed afterwards
Slowing down on the wall below a threshold	The player may attempt to turn around on the wall, or slow down, and it makes sense for this to mean that they can no longer hold onto the wall.	The player is pushed away from the wall	Pass	

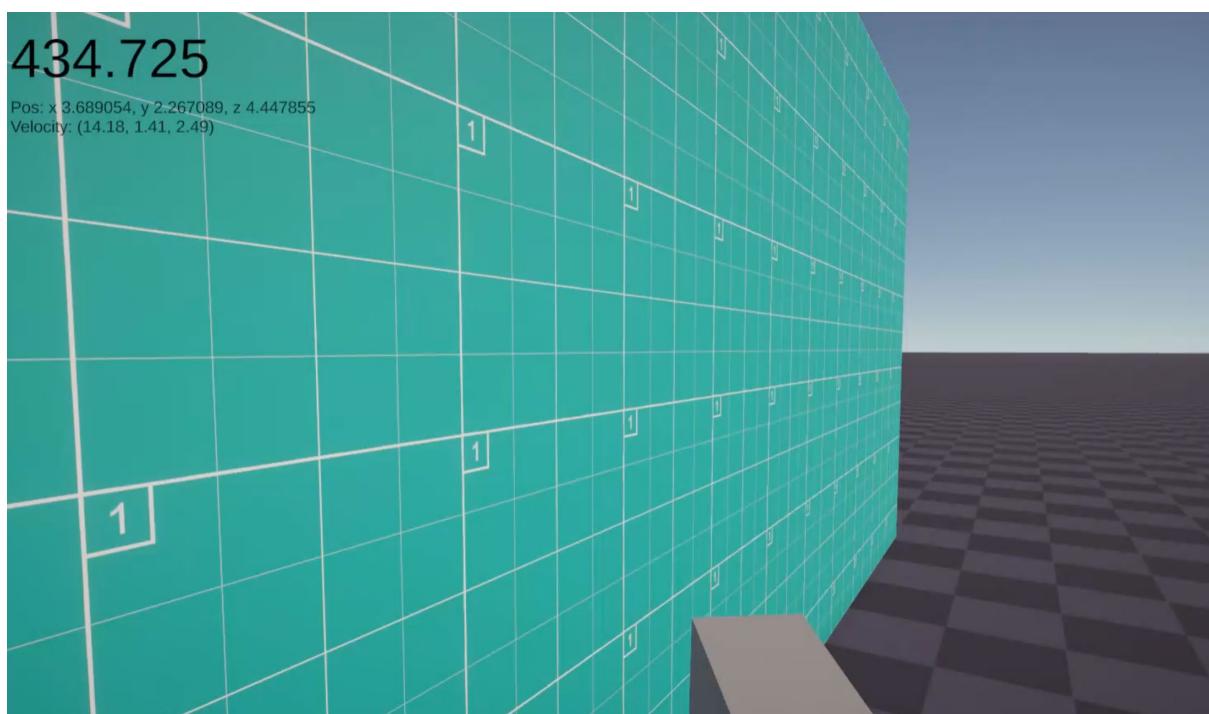
Pressing the jump key in the air while moving downwards on the wall	The player is still affected by gravity, if only by a little bit, so the player can end up moving downwards, which may make the jump useless. Hence this needs to be counteracted	The player will dismount from the wall and gain some speed in the direction of the wall normal and upwards	Pass	Once again, a lack of feedback makes it hard to distinguish between an air jump or a wall jump
Pressing another key (e.g. M)	An input not related to the system should not affect it	The player will slow down on the wall due to friction	Pass	



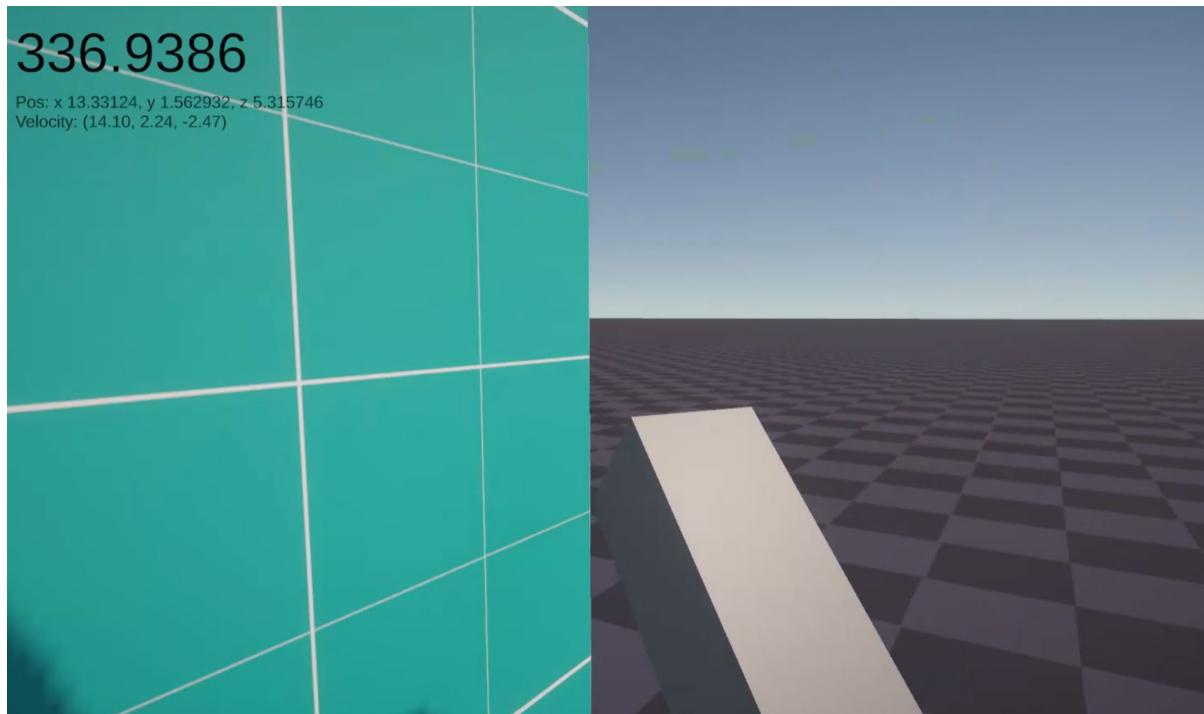
(The player starts on the ground)



(The player jumps onto one wall and runs along it)



(The player jumps off the wall, aiming for the next one)



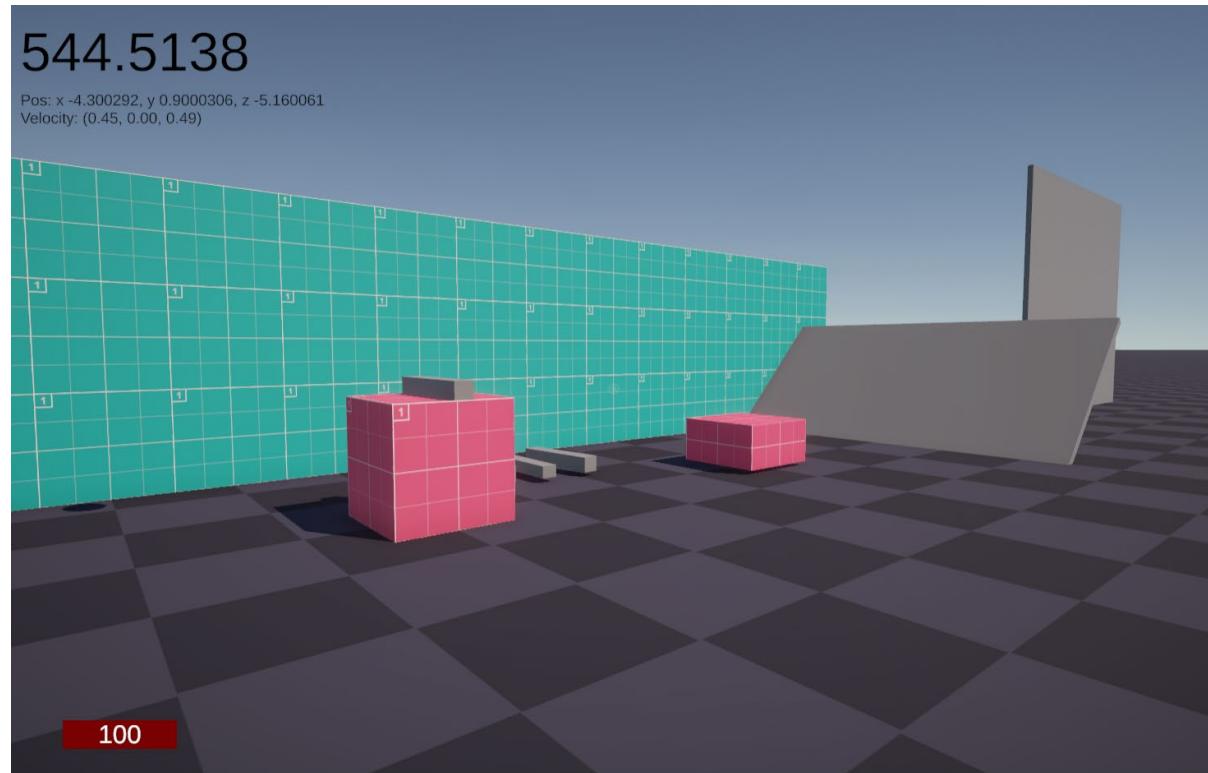
*(The player jumps off the next wall)*

## Using weapons

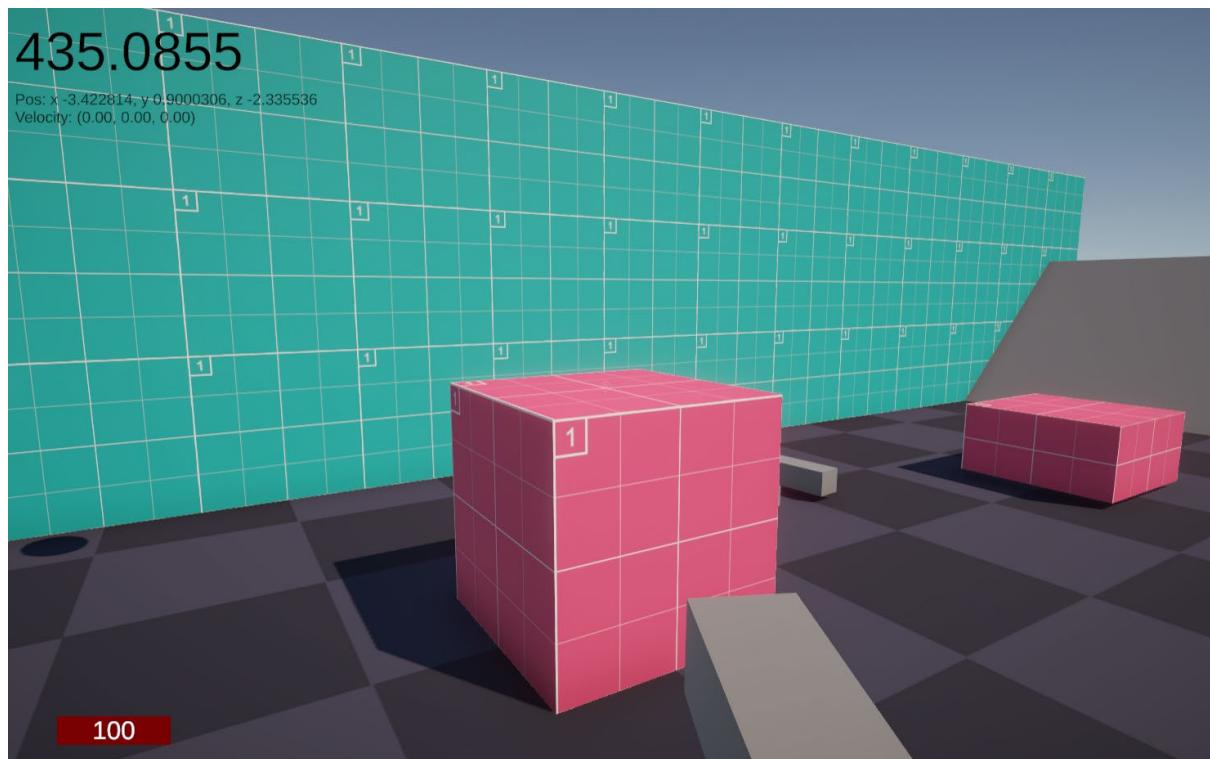
What data fits criteria	Justification	Expected	Pass/Partial/Fail	Comments
Pressing the fire input with an equipped weapon	This is the key the player will press when they want to fire the weapon, and so should be tested correctly	The equipped weapon will fire a bullet	Pass	
Pressing the fire input with no equipped weapon	The player may accidentally press this input, and as such this needs to be handled	Nothing will happen to the weapons	Pass	
Pressing the swap weapon input with no weapons while facing a weapon	The player will need to do this at the start of the level, and so nothing should break when they do so	The weapon will become equipped and move to the player's hand and be added to the list of weapons	Pass	
Pressing the swap weapon	The player may do this when	The weapon will become	Pass	

input with some weapons while facing a weapon	they want to pick up more weapons	equipped and move to the player's hand and be added to the list of weapons		
Pressing the swap weapon input with as many weapons as they can hold while facing a weapon	The player may do this when they run out of ammo on their current weapon	The weapon will become equipped and replace the previously equipped weapon	Pass	
Pressing the swap weapon input while not facing a weapon	The player may accidentally press this input, and as such this needs to be handled	Nothing happens to the weapons	Pass	
Pressing the reload input with an equipped weapon	The player may want to reload their weapons early while they have cover and some breathing time	The currently equipped weapon reloads as much ammunition as it can	Pass	There is no visual feedback that this is the case
Pressing the reload input with an equipped weapon, but no ammunition remaining	The player may want to reload their weapons earlier, but they shouldn't be able to in this case	Nothing happens to the weapons	Pass	There is no visual feedback that the weapon is empty
Pressing the reload input with no equipped weapon	The player may accidentally press this input, and as such this needs to be handled	Nothing happens to the weapons	Pass	
Pressing the zoom input with an equipped weapon, then releasing it	The player will need to zoom in when they want to have a precise shot, and so this should be tested properly	The player's FOV zooms in, then back out.	Pass	

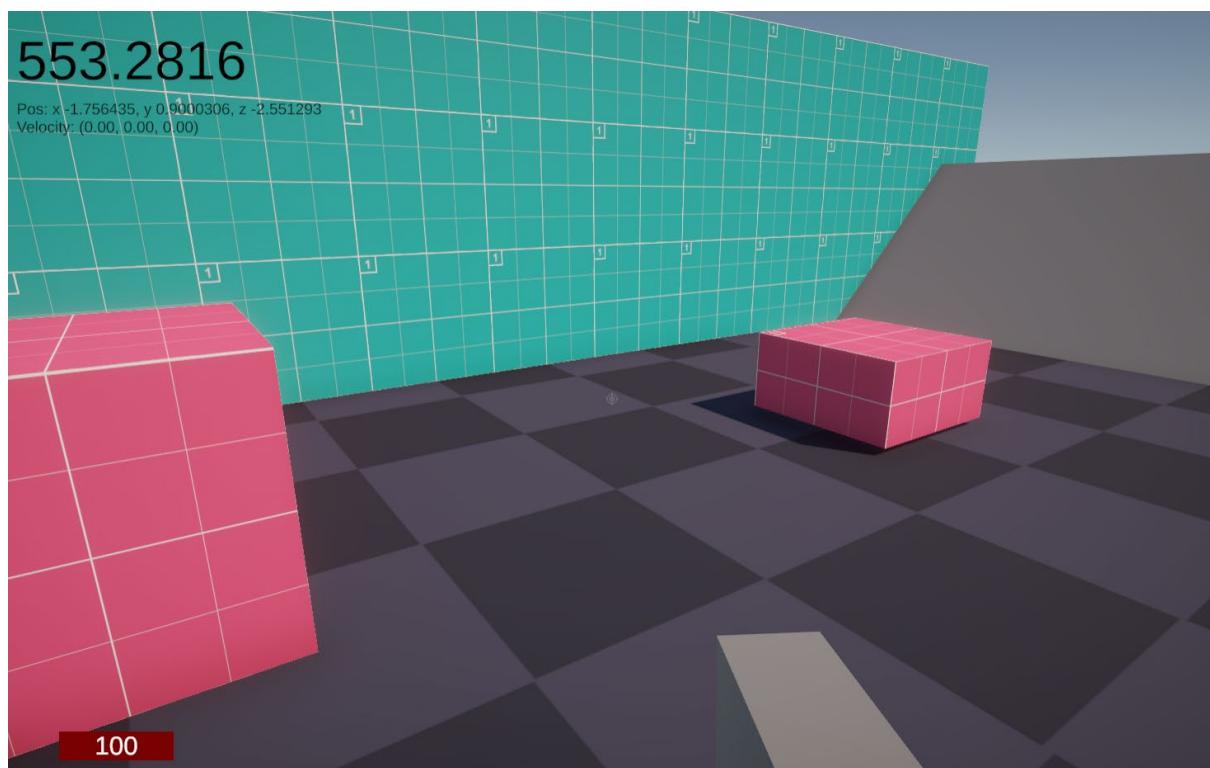
Pressing the zoom input with no equipped weapon	The player may accidentally press this input, and as such this needs to be handled	Nothing happens to the weapons	Pass	
Pressing no input	Testing for the lack of inputs is also something that can happen on any frame, and must be tested correctly	Nothing happens to the weapons	Pass	
Pressing an unrelated key (e.g. M)	This is invalid and should not affect the jumping system, and must be dealt properly	Nothing happens to the weapons	Pass	



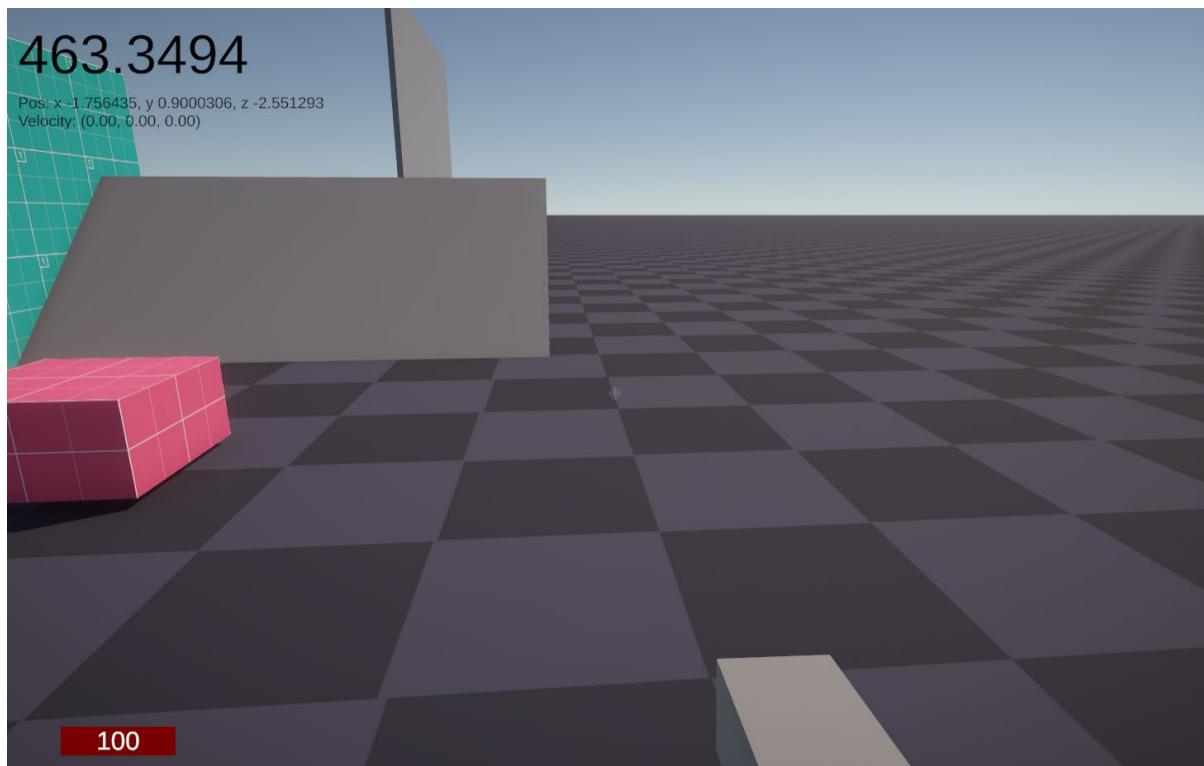
(The player starts with no weapon equipped)



(They then interact and pick up the weapon)



(They then interact again to pick up another 3 weapons)



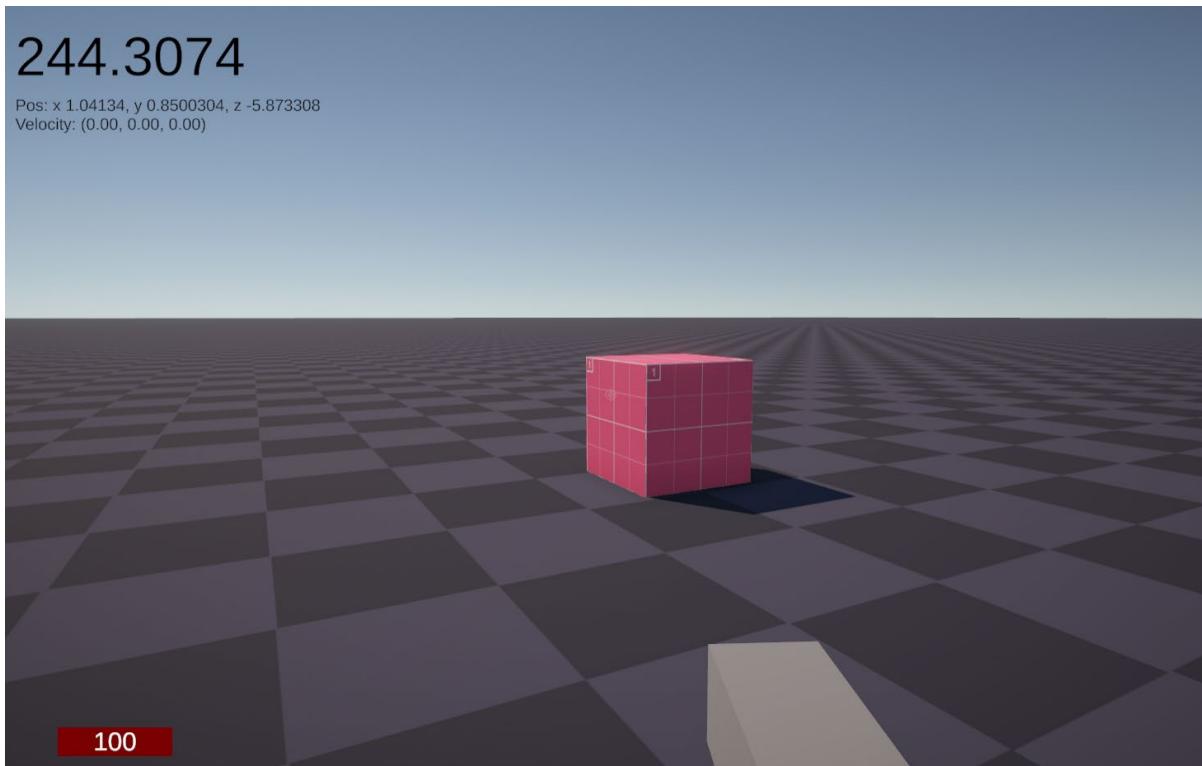
*(The player then swaps to the first slot by scrolling)*

### Taking damage

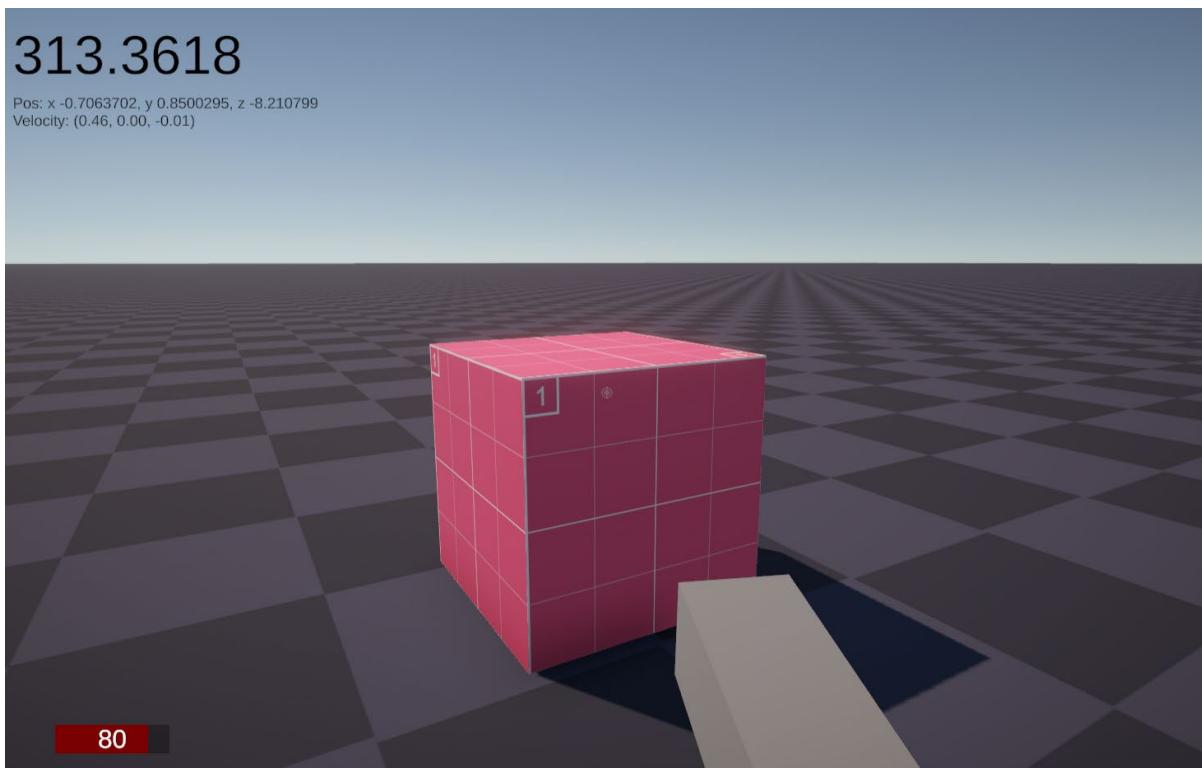
These tests will be evaluated using the spike, which will also simultaneously verify that the spike, and hence environmental damage, works as expected.

What data fits criteria	Justification	Expected	Pass/Partial/Fail	Comments
Get hit	This should lower the player's health, but not enough to kill them, so it should still be visible to the player	The player losses health as shown on the HUD	Pass	
Getting hit enough to lose all of their health	This will cause the player to die, and tests the death screen as well	The player is unable to control their character, and the death screen shows up	Pass	
Pressing 'Load' on the death screen	The player will most definitely want to do this when they die,	The game will reload the player's save data and the	Pass	

	and should be tested properly	player will be able to play once again		
--	-------------------------------	--	--	--



(The player starts at full health)



(The player touches the spike and takes 20 damage)

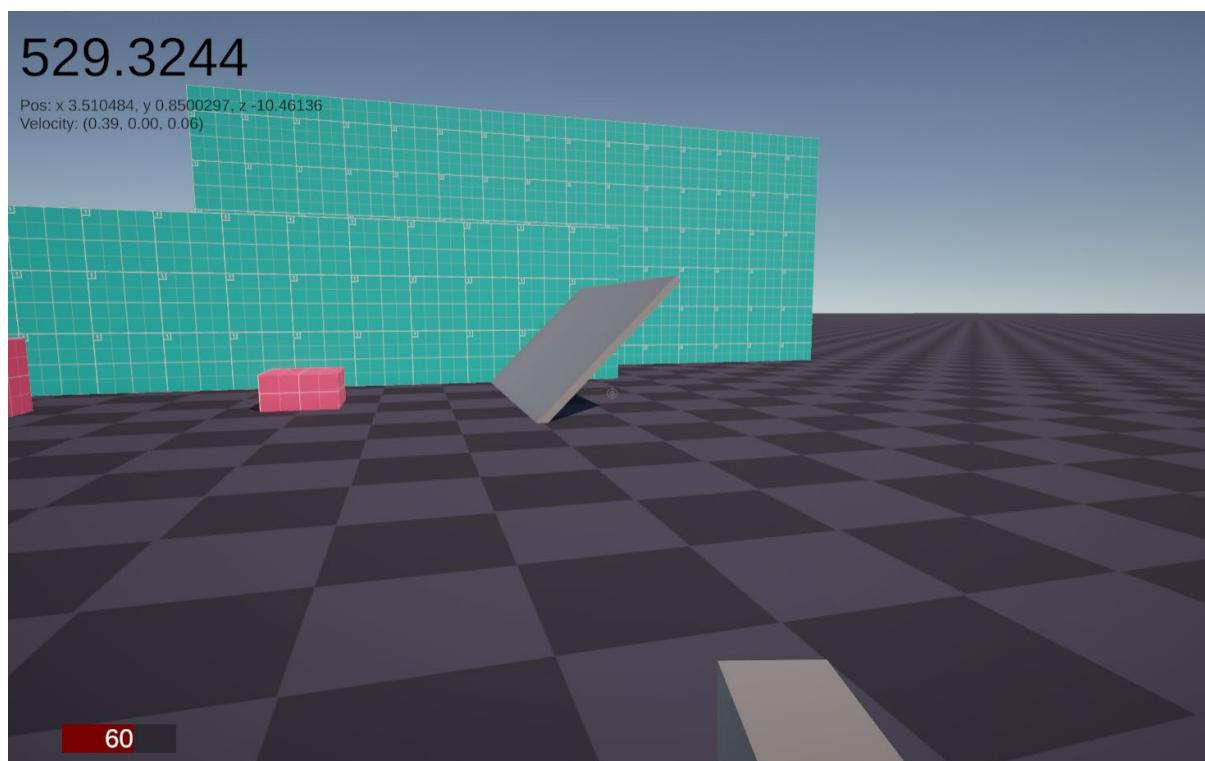


(After touching the spike a few times, the player has died)

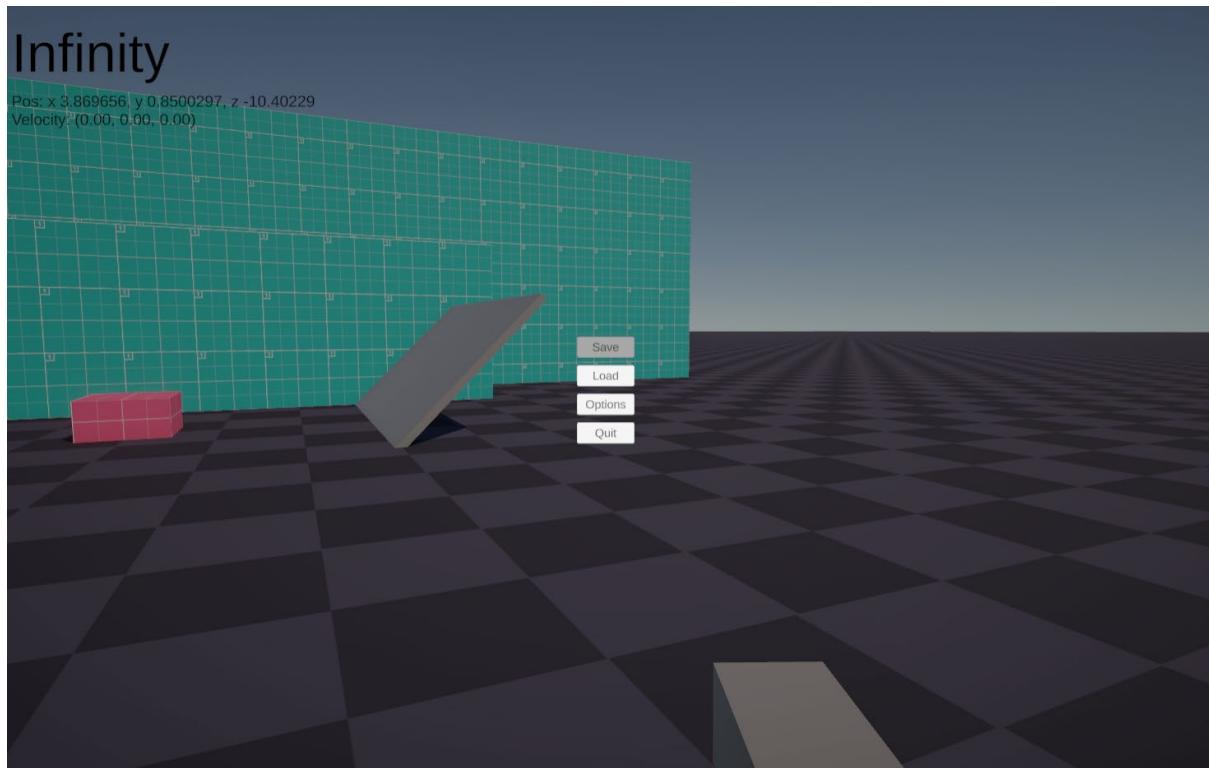
## Saving

What data fits criteria	Justification	Expected	Pass/Partial/Fail	Comments
Pressing any save button	The player will want to save their current progress and not lose it, and this should work well	A new save file is created on the player's computer	Pass	
Making a new save slot	The player may want to replay the game at some point completely fresh, and this should allow them to make completely fresh experience	A new save slot folder is created on the player's computer, along with a new basic save	Pass	
Pressing any load button	The player will most definitely want to load their save when	The game will reload the player's save data and the	Pass	

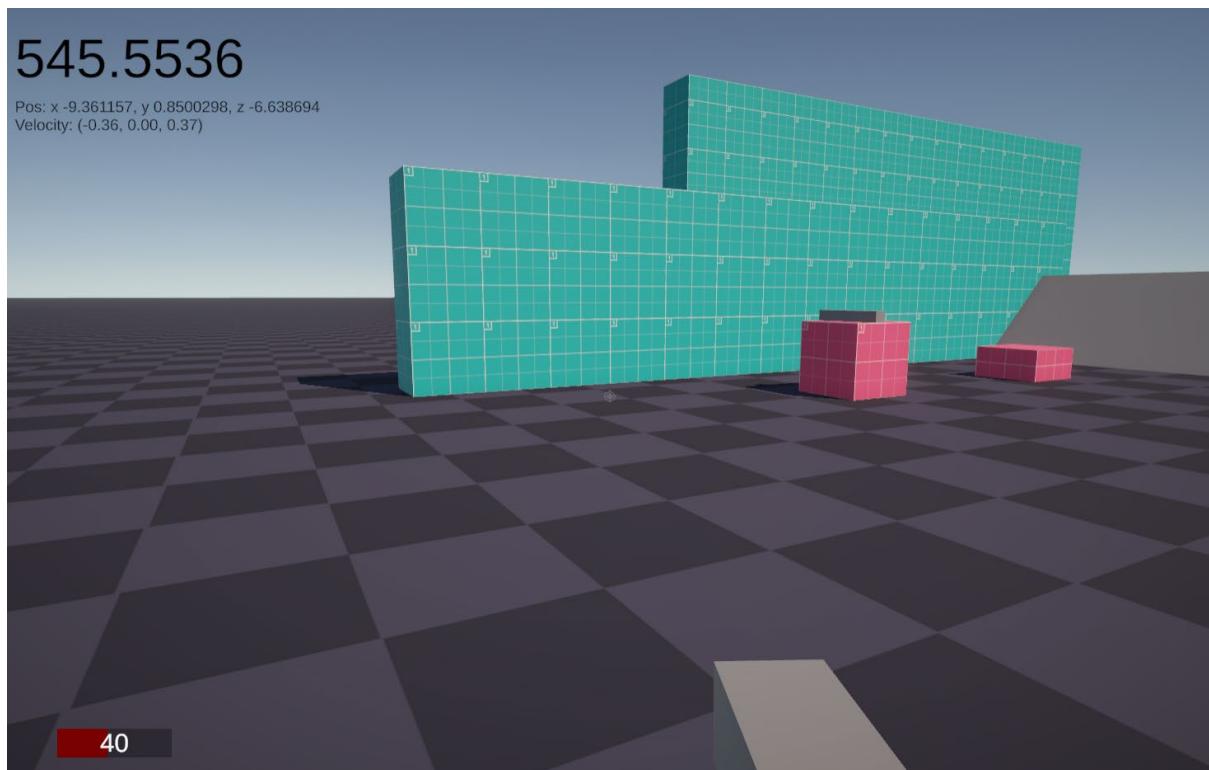
	they get stuck or in other situations	player will be able to play once again		
Selecting to load a slot in the slot menu	The player will want to load into a save slot when they open the game and so this must be handled properly	The game will load the save with a loading screen, and load into the correct level	Pass	



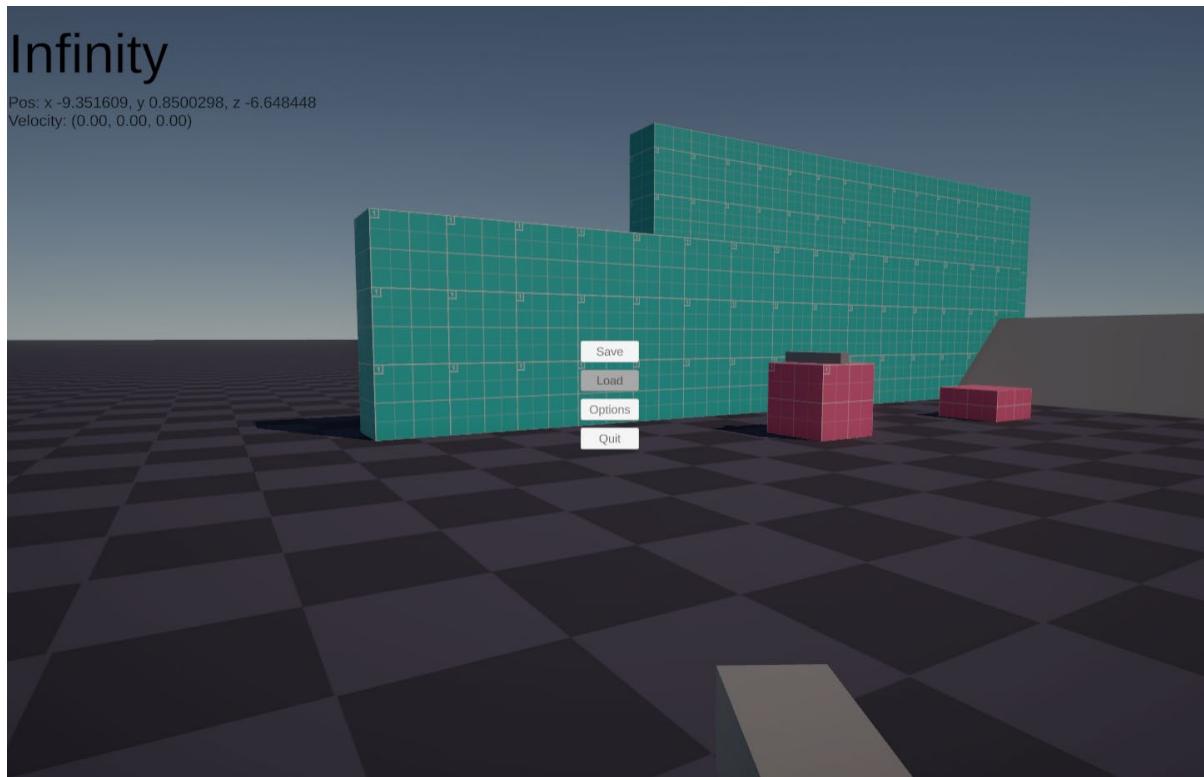
(The player begins at this position with 60 health)



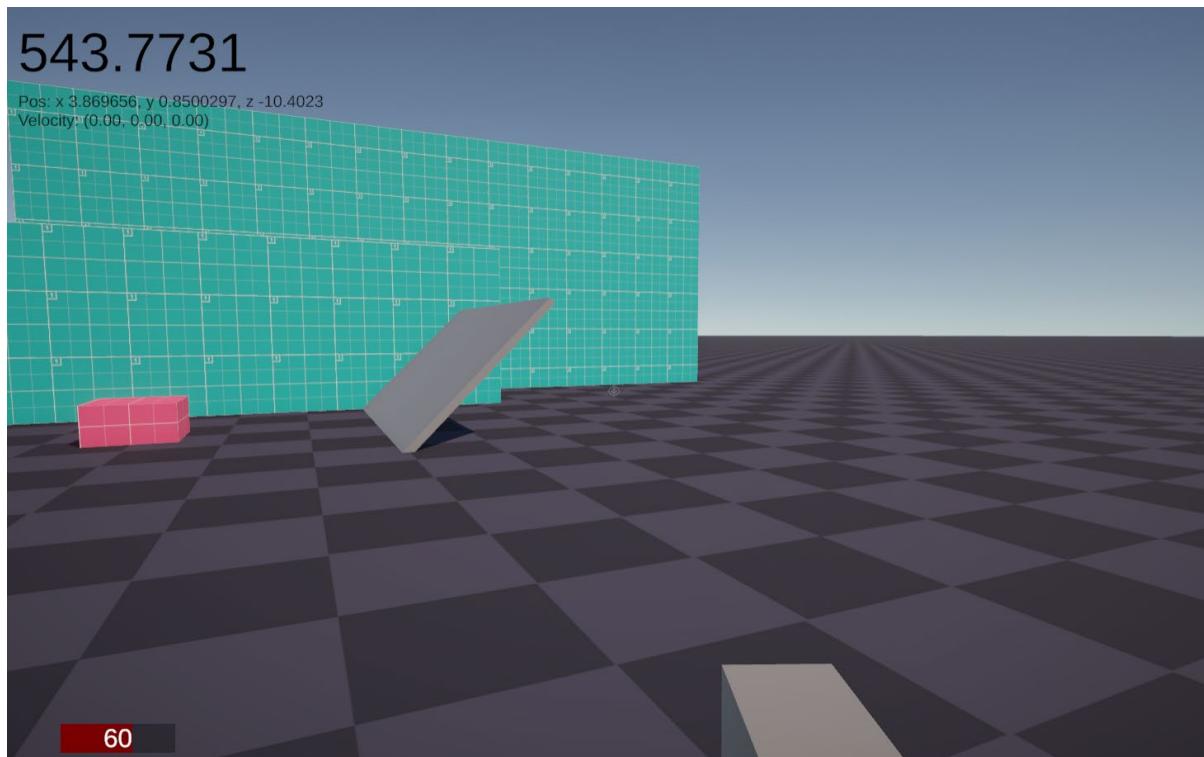
(They then save)



(They move elsewhere and take damage)



(They then load the game)



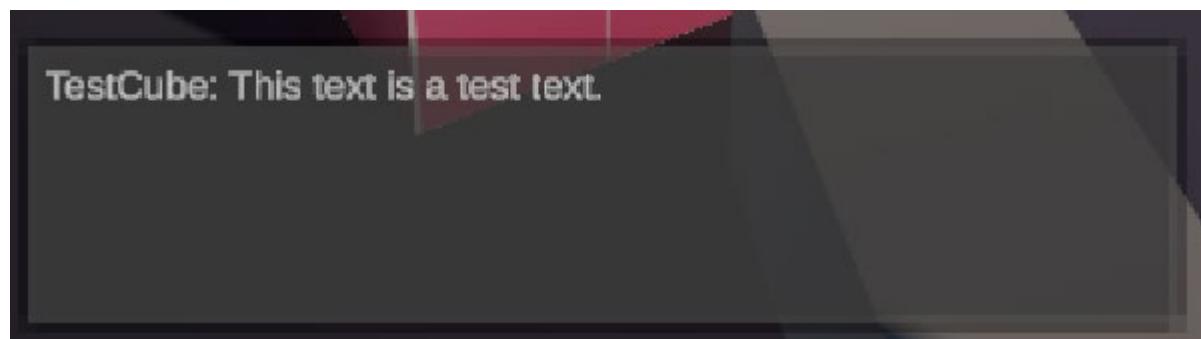
(The save has loaded)

## Dialogue

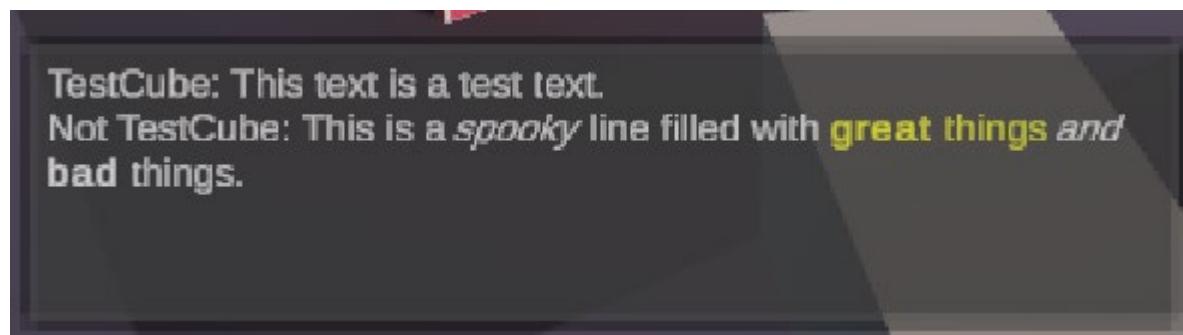
These tests will use an extremely simple test dialogue file that consists of tests to simply interpret lines and actors, along with a simple if statement checking the state of a variable in the save data, ‘seenTutorial’.

What data fits criteria	Justification	Expected	Pass/Partial/Fail	Comments
Pressing the interact key when facing a dialogue object	This is the key the player will press when they want to interact with the dialogue, and so should be tested correctly	A new dialogue line is displayed in the subtitle box that shows up	Pass	
Pressing the interact key when not facing a dialogue object	The system should consider this an invalid input since the input should mean nothing when not facing an interactable	Nothing happens	Pass	
Pressing the interact key when the next line is an if statement that passes	This will test whether the system can handle reading flags	The variable is read, and the line number jumps to the section	Pass	
Pressing the interact key when the next line is an if statement that fails	This will test whether the system can handle reading flags	Only the line number increments	Partial	Will sometimes fail when the next line was a dialogue section
Pressing the interact key when the next line is an assignment statement	This will test whether the system can handle setting flags	The variable is set in the save data flags	Pass	There is no visual feedback for this succeeding
Pressing the interact key when the next line is a choice statement	This will test whether the system can handle choices properly, and respond	The choice menu shows up with each choice available to the player in a grid pattern	Fail	There was no choice menu, and the interpreter reported nothing when

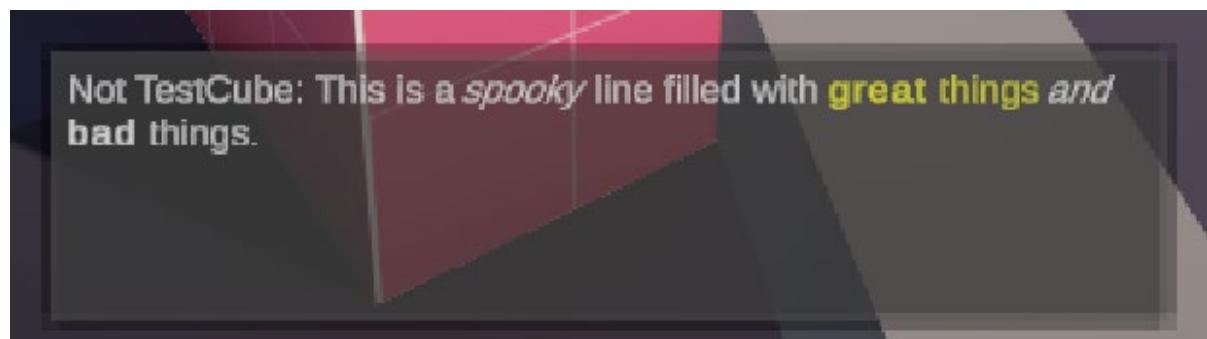
	appropriately			it reached here
Pressing no input	Testing for the lack of inputs is also something that can happen on any frame, and must be tested correctly	The player doesn't move	Pass	
Pressing an unrelated key (e.g. M)	This is invalid and should not affect the dialogue system, and must be dealt properly	Nothing happens	Pass	



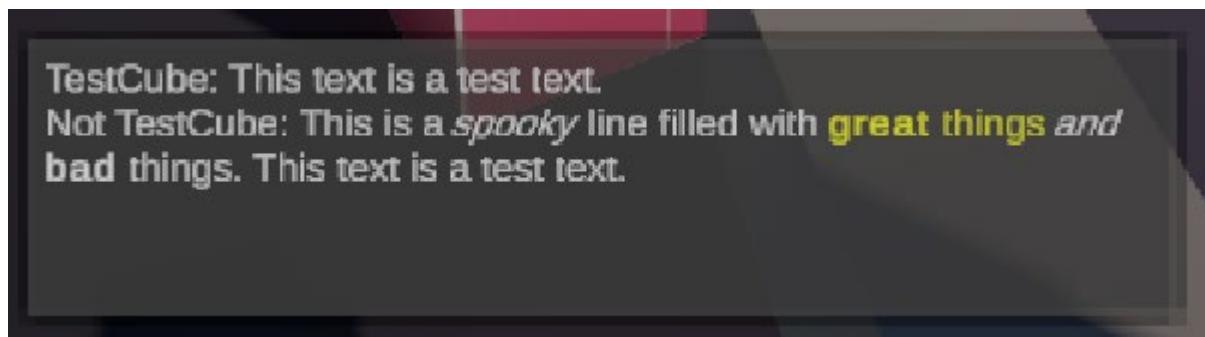
(Interacting with the dialogue interaction once)



(Interacting with the dialogue interaction twice rapidly)



(Not interacting with the dialogue after a bit causes the first one to disappear)



(The first line is also part of the second line and is combined together – This matches the line #A1::#L1#L2 in the .dlg file)

### Entity behaviour

What data fits criteria	Justification	Likert Scale	Comments
The enemy successfully searches and finds me	The enemies will need to search for the player, which will make the experience better for the player, since they can attempt to avoid the sight of the enemy	Disagree	It was very difficult for the enemy to find the player as the entity was not searching near the player
The enemy successfully targets and chases me	The enemy needs to be able to turn towards and chase after the player to be a potential threat to them.	Strongly agree	It is very difficult to escape the entities grasp once the player has been seen, which might be too difficult
Once targeted, the enemy successfully fires at me.	For the enemy to be a threat to the player, and encourage the intended behaviour, it needs to fight against the player	Neutral	The enemy would rarely fire back at the player as it struggled to appropriately fire its weapon when facing the player – the weapon looking slightly off from the player, even when the player was still.
The friendly entities follow the player	Friendly entities are useful to the player	Strongly disagree	There were no friendly entities

successfully	only when they are able to reach the same destination as the player, and this requires that they follow the player		to be seen
The neutral entities run away successfully	Entities that cannot fight should constantly attempt to run from any opposition, signalling to the player that they are not to be fought	Strongly disagree	There were no neutral entities to be seen
The friendly entities help the player when fighting	The purpose of the friendly entities is to fight alongside the player, which needs them to fire against the enemies	Strongly disagree	There were no friendly entities to be seen

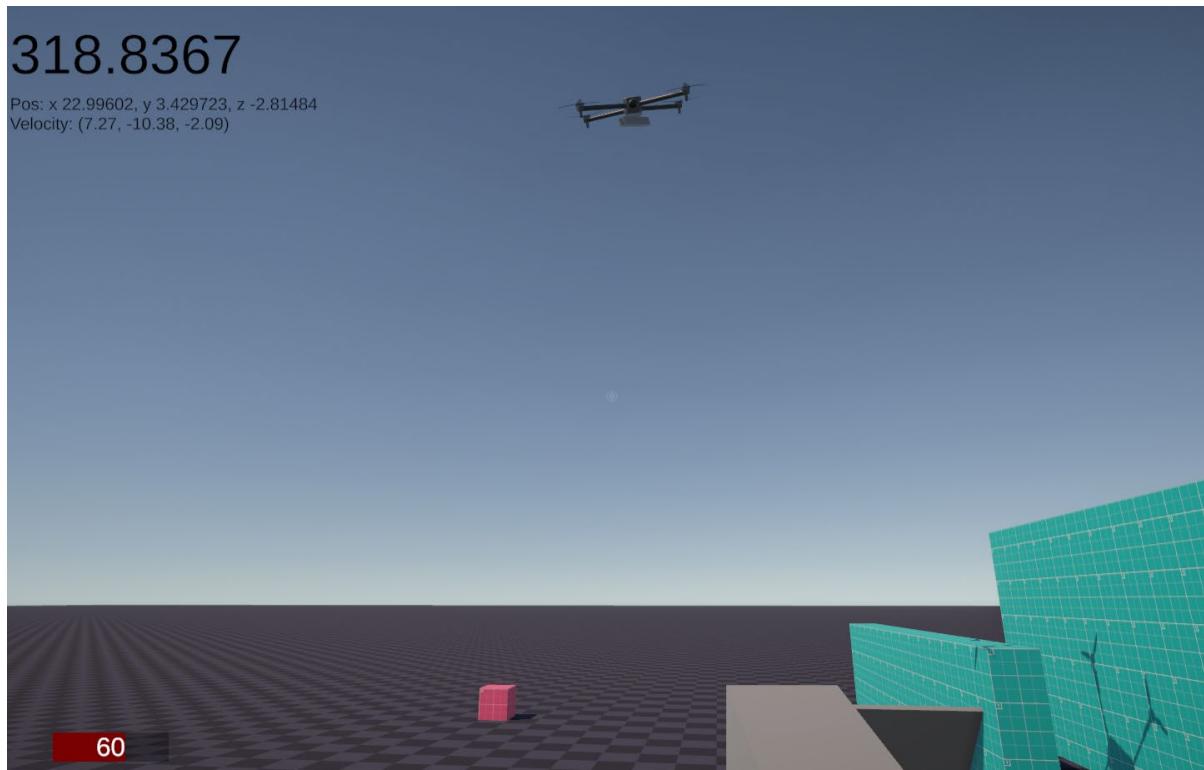
436.9865

Pos: x -4.189445, y 0.8500294, z -22.74584  
 Velocity: (0.00, 0.00, 0.00)



60

(The drone is searching for the player)



(The drone has seen the player and locked onto them)



(The drone has followed and fired at the player and the player has taken damage)

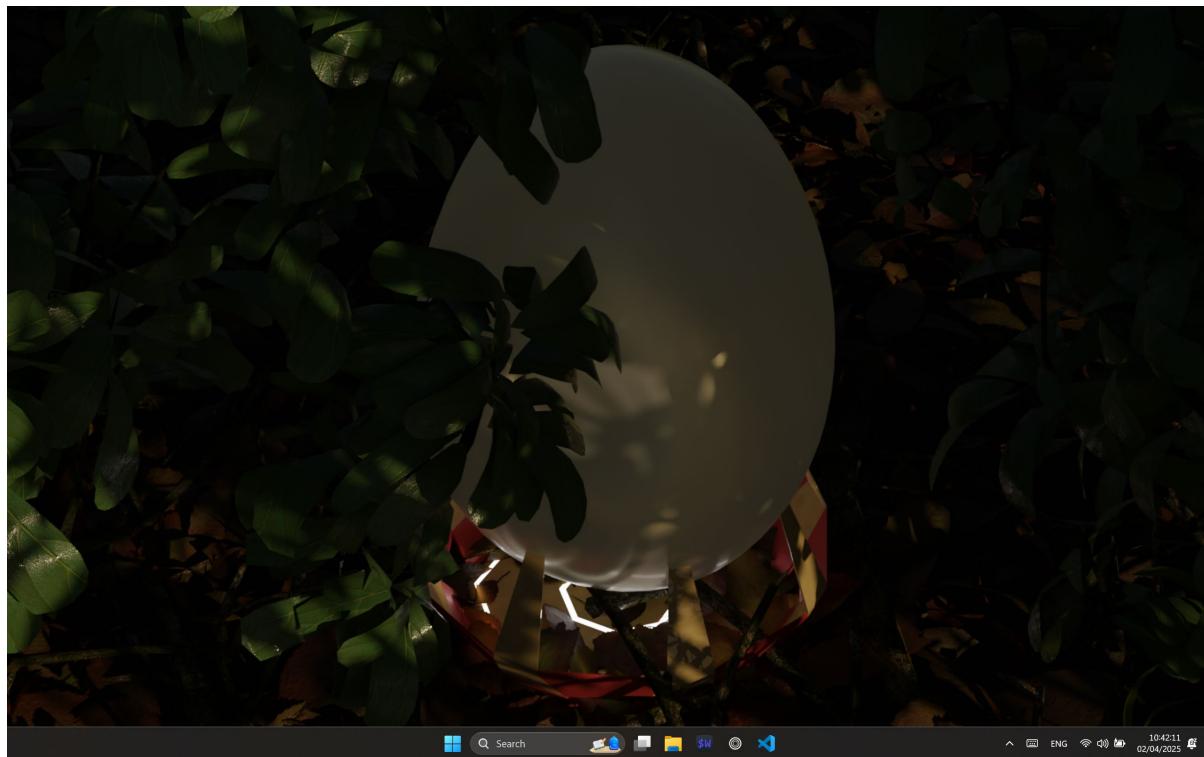
## Usability Tests

Statement	Justification	Likert Scale	Comments
-----------	---------------	--------------	----------

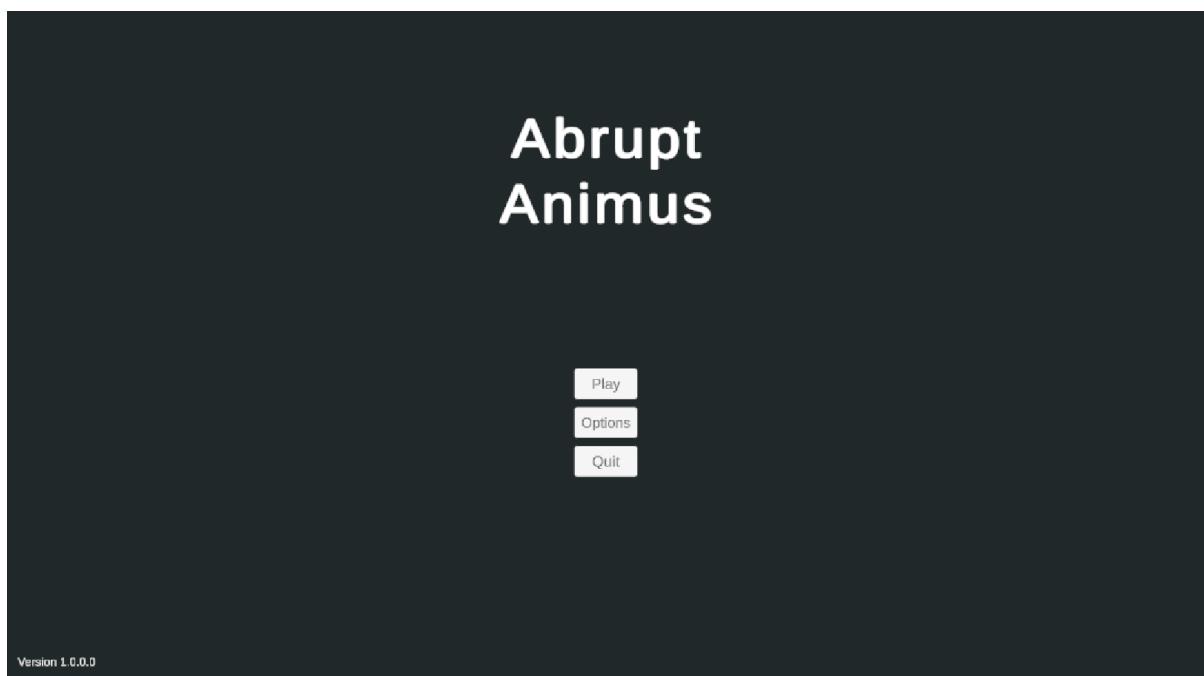
The main menu is easy to navigate	This is the first screen that player will see when they open the game, and so it should be as easy to use as possible	Strongly agree	Very little to click on so it was easy to see where to go
The slot menu is easy to navigate	This is one of the more important menus that the player may interact with and as such it should not be difficult to navigate	Agree	When the player has lots of slots, it can be tedious and difficult to find the right slot through only the slot number
The pause menu is easy to navigate	The player will have the ability to do quite a lot during the game while accessing this menu, so anything that makes this menu difficult could lead to a frustrating experience	Strongly agree	
The death menu is easy to navigate	The player is likely to visit this menu quite often, and so it should be as simple as possible to allow them to leave it as fast as possible	Strongly agree	
The options menu is easy to navigate	The player will have a lot of control within this menu and so this menu should be usable and simple to use	Neutral	Having no values on the sliders made it difficult to tell if the same value again if changed
I can see my health on the HUD clearly.	The player should be able to see their health easily since this is an important part of the game and the player may be in danger if they can't see it	Agree	When looking at somewhere white, the text becomes unreadable as it blends in
The health does not obstruct my view of the game	Conversely, if the health is too large, then the player will not be able to see the	Agree	The health bar is opaque and does cover the small screen space

	game's world and may be in more danger		that it does cover
I can see my current weapon on the HUD	The player should be able to see what weapon they have equipped and any information about that weapon	Strongly disagree	This was completely missing from the HUD
I can see the previous and next weapons on the HUD	The player should also be able to see what weapons they may be changing too as this also important information	Strongly disagree	This was also completely missing from the HUD

*Evidence – There is no game*

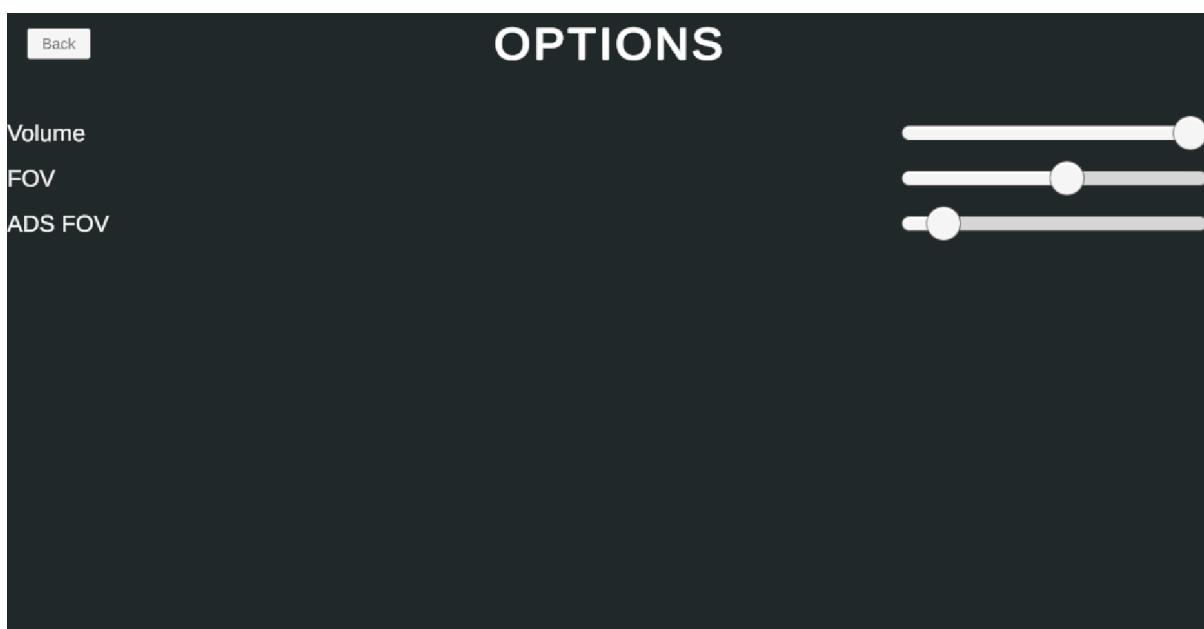


*Evidence – Opening the game, Main Menu*

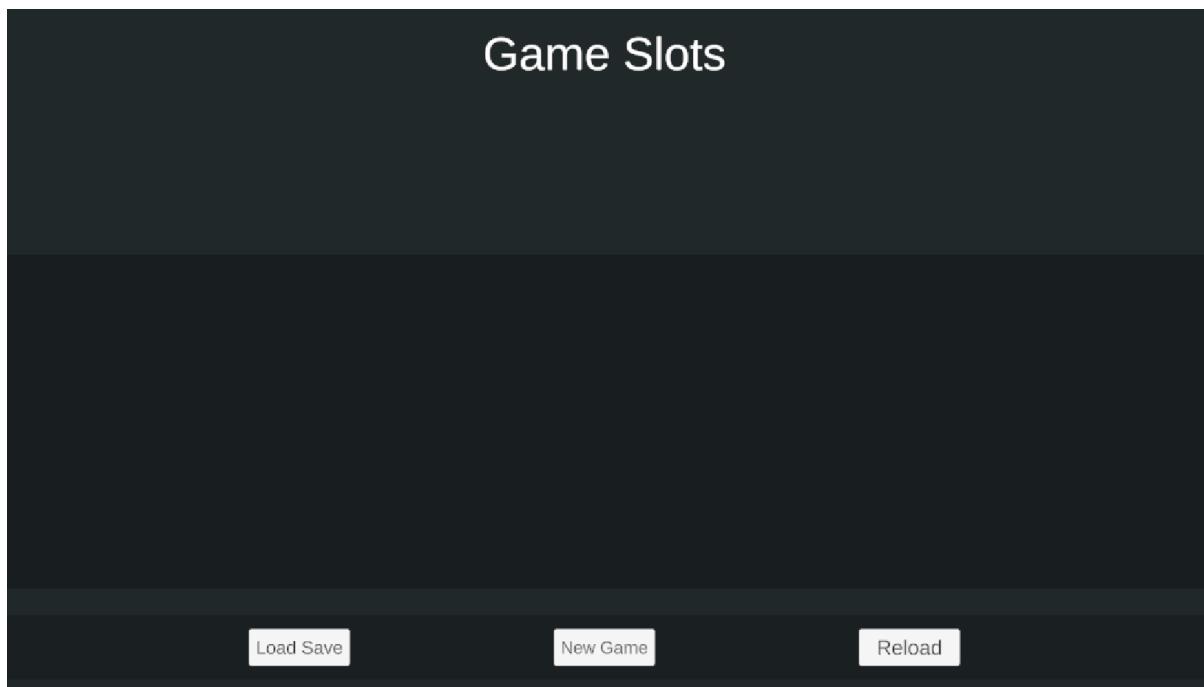


*(What the player sees as they first open the game)*

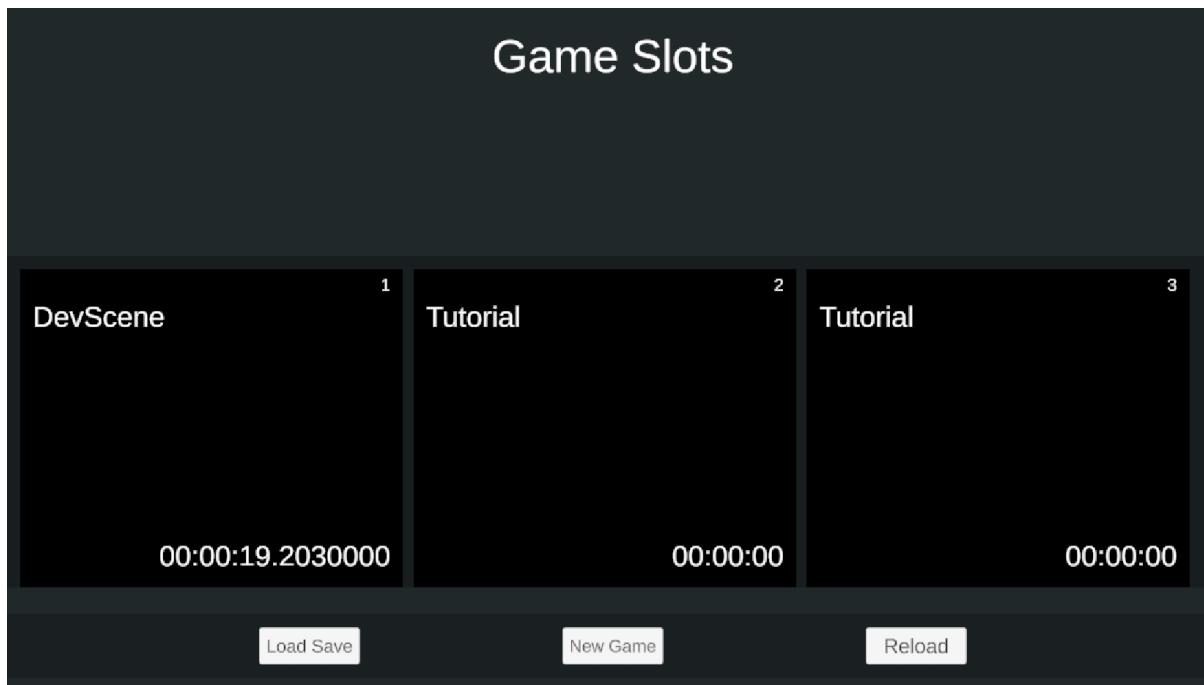
*Evidence – Option Menu in the Main Menu*



*(The options menu in the main menu)*

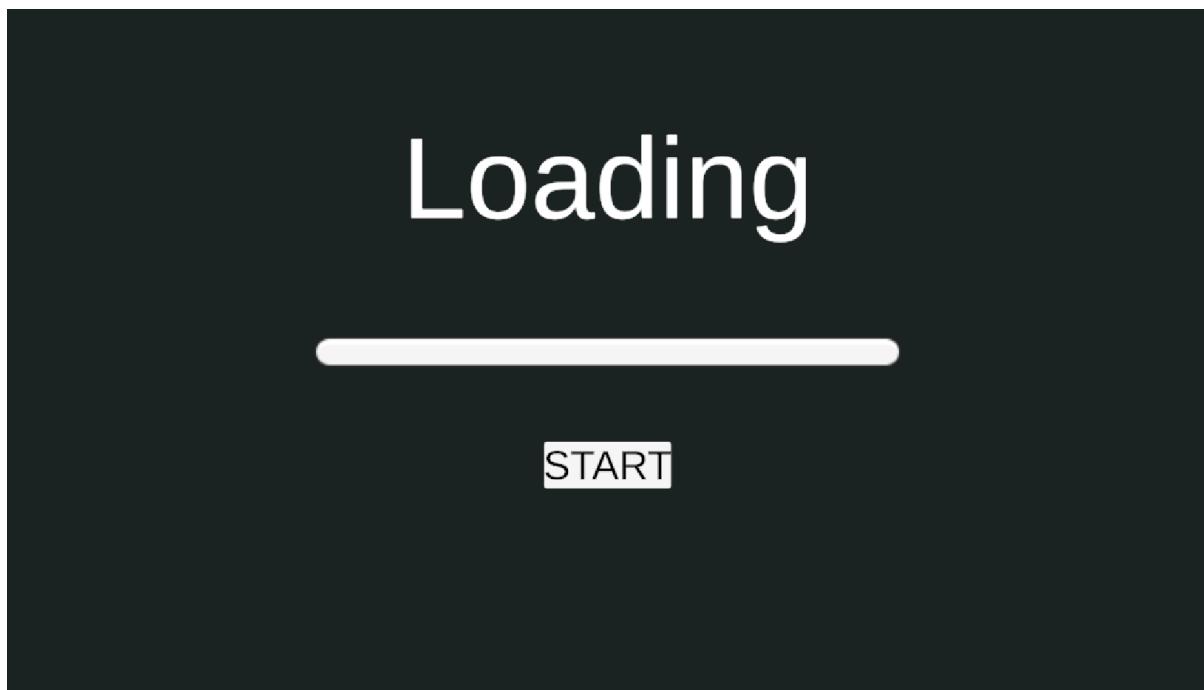
*Evidence – Empty Slot Menu*

(The slots menu upon opening the game for the first time)

*Evidence – Filled Slot Menu*

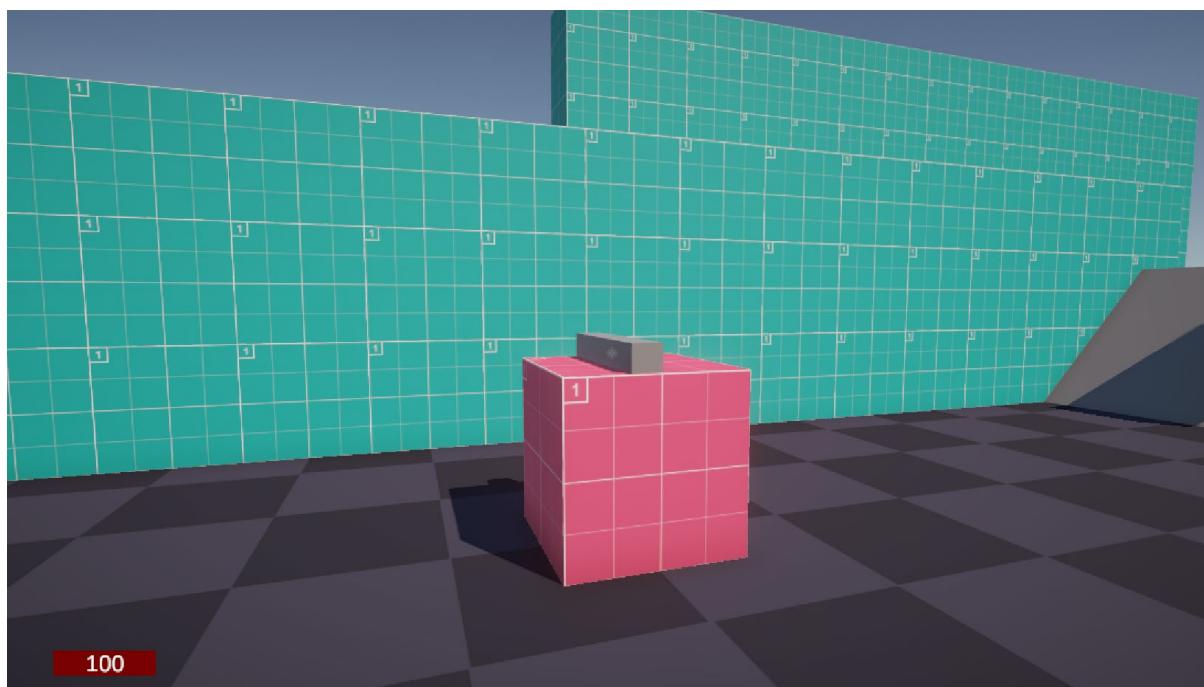
(The slots menu with some filled in slots)

*Evidence – Loading Screen*



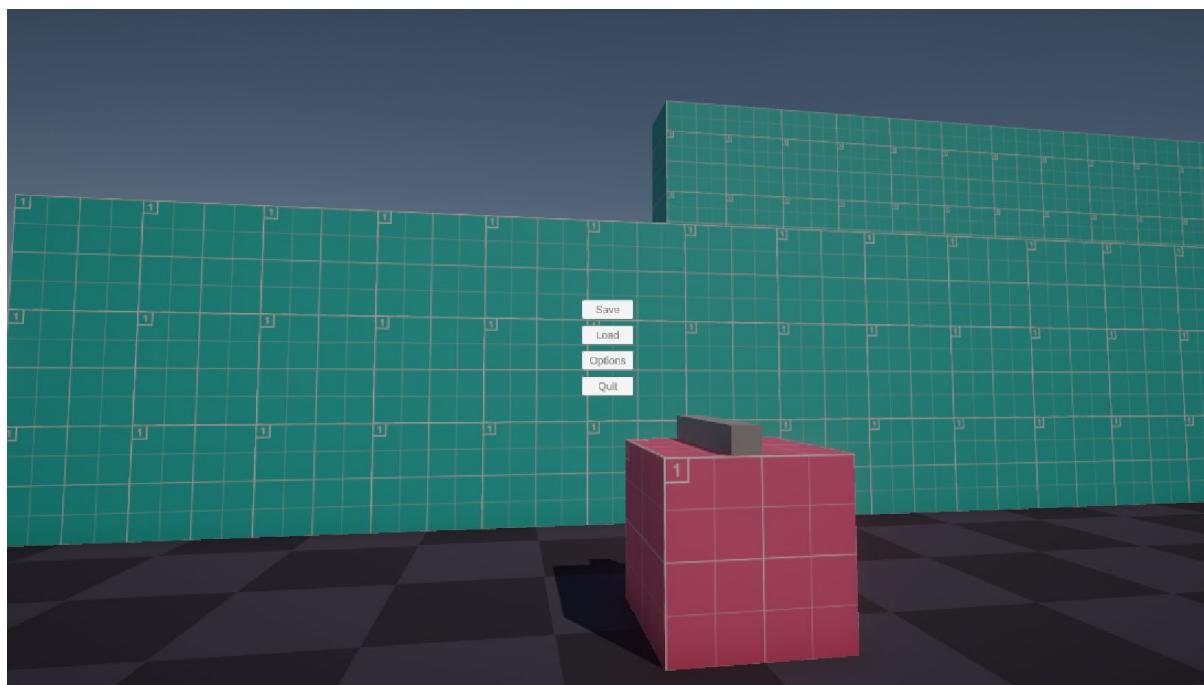
(The loading screen, with the scene finished loading)

*Evidence – HUD*



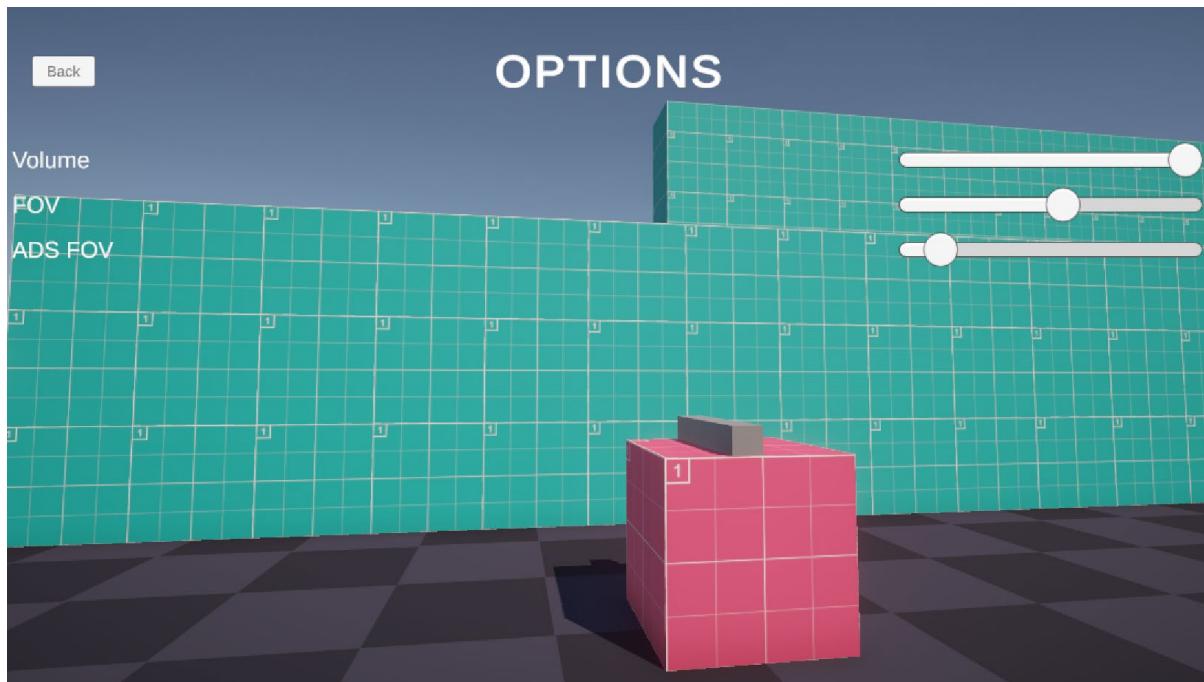
(The HUD and player view when the player first loads into a save)

*Evidence – Pause Menu*



(The pause menu)

*Evidence – Options menu in the Pause Menu*



(The options menu in the pause menu)

## Success Criteria

Each criterion will be evaluated based upon the above test data and whether they were met fully, partially, or not met at all. They will also include extra comments from both stakeholders.

Criterion	Met rating	Stakeholder Comments
Strafing Movement	Fully met	Some of the movement feels a little bit slippery, and there is also the jittering bug that is experienced when no input is held after the player has moved.
Jump	Fully met	Double jumping works well but could add an extra boost in the direction that the player is facing.
Crouching	Fully met	
Firing, swapping, reloading, zooming different weapons	Partially met	While the player could hold multiple weapons, these were all the same type, as different types of weapons were not implemented.
Can take damage	Fully met	
Ability to go back to a save on death	Fully met	It may be useful to have information on how the player could avoid dying the way they died on this screen as a future feature.
Saving	Fully met	It could also be useful for the player to see all of the saves inside of a particular save slot, although this feature could add unnecessary bloat to the player's screen.
Loading	Partially met	Excluding the known error with the loading of a save while in the pause menu, this system could allow the player to load one of the saves the player inside of their save slot, like the above note.
Interactable menus with audio feedback	Fully met	
Dialogue	Partially met	The ability for the player to have choices with the system was cut.

Enemies searching and fighting back	Partially met	Enemy searching seems to be inconsistent and unreliable, especially for the drone. The enemy also struggles to point its weapon at the player, although once it does, it fires adequately.
The game runs at a minimum of 30 fps on typical hardware, as according to the Steam Hardware Survey	Fully met	The game runs far above 30 fps, even when stressed with extra enemies.
The game will have as few glitches as possible, with absolutely no game-breaking ones	Fully met	No game-breaking bugs were found

## Interview

Henry Masters: Overall, I consider this project a success, even though there were some features that were unable to be implemented effectively, or at all. This has demonstrated the concept for this game very well, and my team have decided to continue developing this game as a result.

Amarveer Flora: I would also like to note that there are very few bugs with the game overall, so many of the features that are already in the game will not need much tweaking for you to use it.

Henry Masters: Ah, thank you for that extra note!

Stakeholder Signature:



## Limitations and Maintenance

Due to the extremely tight constraint of time, many planned features were not implemented, along with potential improvements that could have been put in. Of

features that were not implemented at all, these include the end of level scoring system, along with optional special movement type – grappling. These systems could have been added, or like the grappling mechanic, identified too be as too much earlier on in the project design, rather than never reaching them in development. However, moving forward, implementing these systems can utilise the other systems already put into place, adding into them as necessary, and hopefully not requiring much modification of the already written systems. To implement grappling, I could add this into the movement script, just like wallrunning, or I could add it through a branched off script that uses a separate script that communicates with the player's manager. Both ways will need to be checked through to see which one would be better to do in the long-term development for the game. In terms of the scoring system, this should be a completely isolated system, with most of its influence on other systems already having been implemented correctly. For example, its influence in the save system has already been partly implemented in the form of a template for what the data should look like. I could then further develop the system to modify and apply this data appropriately within the game manager.

There were also parts of the game that had to be reduced in size, such as the different types of weapons. I could add in melee weapons, which allows a different style of play for the player. It is important to allow the player to have different playstyles to encourage the player to use everything that they have to their disposal, and experiment with what I have handed to them. It will also allow more players to enjoy the game and give players their own artificial challenge by attempting to play without a type of weapon. These melee weapons could use a different system for applying damage, such as using part of the spike script to apply damage on contact with the weapon's collider. On the other hand, a different, also common type of weapon is the explosive, which will encourage a completely different type of play style to both typical guns and melee weapons. Explosive weapons will also need to be balanced differently to other weapons, to make sure that it doesn't become too powerful and have a negative effect on the player's experience. I could do this by making explosives have a high damage effect on enemies within a certain range of them but conversely having an extremely long reload time or a slow rate of fire.

Another part of the combat system is the entity system. According to the evaluative tests, the entity behaviour is definitely dodgy and requires much improvement. Firstly, the entities are unable to sense the player accurately, especially for the drone and whenever the player is slightly above the soldier and tank. This is due to the entities' searching algorithm using rays and only shooting them out in 1 plane – the flat XZ plane. This means that it never searches slightly above or below the entity and hence misses out the majority of the positions the player could be in, and where the player would expect that the entity would see them. This effects the player's experience since it defeats the entire point of the enemies, making there be hardly any difficulty in the

game, and taking a lot of the fun out of it. Firing more rays in the varied angles is a potential solution to this but falls into a performance problem when many enemies are firing of many, many rays to make sure that they don't miss the player. A much more performant alternative involves a sphere cast. I could use 1 large sphere for each entity that encompasses their entire detection range and then work out the angle between it and every other entity that it collides with to decide whether it is worth to cast an extra ray to check it should actually be detected by the entity. This, while being more complicated, would completely fix the issues with the entity detection. A single sphere cast is also not that much more difficult to run than a single ray cast, making this the better option to implement moving forward.

Another feature for the entities was the ability for the player to have their own friendly team that would fight with them, and this would require the entity searching algorithm to check for entities with the opposite team as them, and then have the player be part of one of these teams. This would allow a way for the game to manage the difficulty the player experiences much better and allows a story aspect to develop much better in the game, hence justifying this feature. Having a story would lean into the secondary RPG genre of the game, which would justify the genre as being part of the game. With no story, the character is not fulfilling a role, and without any systems for the story exist, any story elements that are added would not mix as well with the game and feel natural. The dialogue system naturally ties into aiding a potential story, and this system was not completed in its entirety. The most interactive element of it, the player's ability to make a choice, was cut out due to time constraints. This also meant that the menu for it was never created, and I could have added this into the game as a future addition, especially if I were to have more time. It would also be useful to fix the issue with the if statement, making sure that the player can fully experience all the options of dialogue that the system supplies.

The next limitation surrounds the optimisation of the game and making it easier to maintain and develop upon by the team at SillyAustralian. This limitation comes down to the duplicated code that exists between 2 functions in the player's movement: Wallrun() and Movement(). Wallrun() uses virtually the exact same code as Movement(), adding the possibility of making it a wrapper function instead, and passing through the changed variable instead. This would improve the ease in maintaining the code and solving any errors in it, which is always beneficial.

The final limitations all surround usability and feedback. It is important that when the player does something, namely an input, they should get a response back, whether it be via particle effects, sound effects, or something else. There are many instances in this game where it would benefit from a response to the player's actions. The first of these instances is constantly experienced by the player – the use of their weapon. When the player presses the fire button, there is no response from the game that the weapon has

been fired, nor whether it hit anything. This can make it very confusing to understand whether the gun is out of ammunition or not. The only feedback that exists for the weapon being fired is when the enemy visibly disappears upon being killed, but this requires that the player has landed all the shots required to kill the enemy, which can frustrate the player. To fix this, I could implement sound effects that are played whenever the player fires the weapon, and also when it is reloading or out of ammunition. However, sourcing good sound effects are extremely difficult, especially due to budget constraint. I could also make the sound effect myself, but for a game meant to be extremely serious, I would be unable to make appropriate clips with my current ability, since I do not know anything about sound design. Another thing that could aid the issues surrounding weapon feedback is the display of the weapon's current information using the intended feature on the HUD. Since this was missing, it became nigh impossible to get any feedback on your weapons' current state. I could add this to the display and link this using an auxiliary script to translate between the HUD and PlayerGunInteraction. This will improve the usability of the game and make it easier to know what is happening in the game.

The last instance of something absolutely needing feedback that was pointed out in the evaluative testing is the wallrunning. It was pointed out that it wasn't always clear when the player had initially started wallrunning, nor when the player had jumped off the wall or had fallen off the wall and then double jumped. For the former, I could implement a camera tilt away from the wall surface to truly make the player feel like they are running on the wall. For the latter, this can be solved with two different sound effects for jumping normally and for jumping, which runs into the same issues for the weapon sound effects. However, nonetheless, this should work well in increasing the usability of the game.

## Bibliography

- Anon., 2024. [Online]  
Available at: <https://tf2sr.github.io/TF2SR-Wiki/>
- Anon., n.d. *How to Tune a PID Controller*. [Online]  
Available at: <https://pidexplained.com/how-to-tune-a-pid-controller/>  
[Accessed 22 November 2024].
- Atwood, J. & Spoisky, J., 2008. *Stack Overflow*. [Online]  
Available at: <https://stackoverflow.com/>  
[Accessed 26 June 2024].
- d'Apprentissage, T. d. F. e., n.d. *Orthogonal Unit Differentiation*. [Online]  
Available at:  
[https://tecfa.unige.ch/tecfa/maltt/VIP/Ressources/Articles/Game%20Design%20Patterns/collection/Alphabetical\\_Patterns/OrthogonalUnitDifferentiation.htm#:~:text=When%20Units%20in%20a%20game,with%20different%20types%20of%20abilities.](https://tecfa.unige.ch/tecfa/maltt/VIP/Ressources/Articles/Game%20Design%20Patterns/collection/Alphabetical_Patterns/OrthogonalUnitDifferentiation.htm#:~:text=When%20Units%20in%20a%20game,with%20different%20types%20of%20abilities.)  
[Accessed 24 July 2024].
- Dev, P., 2023. *Collide And Slide - \*Actually Decent\* Character Collision From Scratch*. [Online]  
Available at: <https://www.youtube.com/watch?v=YR6Q7dUz2uk>  
[Accessed 25 July 2024].
- Electronic Arts Inc.; Respawn Entertainment, 2018. *Titanfall 2*. [Online]  
Available at: <https://www.ea.com/en-gb/games/titanfall/titanfall-2>
- Fadi, E., 2025. *Link to my GitHub*. [Online]  
Available at: <https://github.com/RandomEF/Abrupt-Animus>
- Frei, D., 2022. *Saving Data in Unity using SQLite*, s.l.: s.n.
- GeeksforGeeks, 2008. *Geeks for Geeks*. [Online]  
Available at: <https://www.geeksforgeeks.org/>  
[Accessed 23 September 2024].
- gskinner, n.d. *RegExr*. [Online]  
Available at: <https://regexr.com/>  
[Accessed 27 August 2024].
- Half Life Fandom, 2007. *Overwatch Soldier*. [Online]  
Available at: [https://half-life.fandom.com/wiki/Overwatch\\_Soldier](https://half-life.fandom.com/wiki/Overwatch_Soldier)  
[Accessed 17 September 2024].
- Huertas, R., n.d. s.l.: s.n.

id Software, 2016. *DOOM*. [Online]

Available at: <https://bethesda.net/en/game/doom-2016>

id Software, 2020. *DOOM Eternal*. [Online]

Available at: <https://mods.bethesda.net/doometernal>

Iyer, A., 2018. *Titanfall 2: How Design Informs Speed*, s.l.: s.n.

Leprawel, n.d. *Parkour Controller Script*, s.l.: s.n.

Microsoft, 2024. *C# Language Documentation*. [Online]

Available at: <https://learn.microsoft.com/en-us/dotnet/csharp/>

[Accessed 6 July 2024].

Official, T., 2016. *Titanfall 2: Inside Development*, s.l.: s.n.

OMOCAT LLC., 2020. *Omori*. [Online]

Available at: <https://www.omori-game.com/en>

One More Level; Slipgate Ironworks, 2020. *Ghostrunner*. [Online]

Available at: <https://ghostrunnergame.com/>

Patala, A. ", 2020. *Ultrakill*. [Online]

Available at: <https://store.steampowered.com/app/1229490/ULTRAKILL/>

SFB Games, 2021. *Chiptone*. [Online]

Available at: <https://sfbgames.itch.io/chiptone>

[Accessed 20 February 2025].

Thirlsund, A., 2012. *Brackeys YouTube channel*. [Online]

Available at: <https://www.youtube.com/@Brackeys>

[Accessed 4 September 2024].

Unity Technologies, 2022. *Megacity Metro – Large-Scale Multiplayer Demo | Unity*.

[Online]

Available at: <https://unity.com/demos/megacity-competitive-action-sample>

[Accessed 22 February 2025].

Unity Technologies, 2025. *Unity Documentation*. [Online]

Available at:

<https://docs.unity3d.com/6000.0/Documentation/ScriptReference/index.html>

Valve Corporation, 1998. *Half Life*. [Online]

Available at: <https://www.half-life.com/en/halflife>

Valve Corporation, 2025. *Steam Hardware Survey*. [Online]

Available at: <https://store.steampowered.com/hwsurvey>

[Accessed 28 March 2025].

Valve Developer Community, 2005. *Category:AI - Valve Developer Community*. [Online] Available at: <https://developer.valvesoftware.com/wiki/Category:AI> [Accessed 18 September 2024].

## Appendix (All the game's code)

### UISounds.cs

```
using UnityEngine;
using UnityEngine.Events;
using UnityEngine.AddressableAssets;
using UnityEngine.ResourceManagement.AsyncOperations;
using UnityEngine.UI;

public class UISounds : MonoBehaviour, IPointerEnterHandler
{
    [SerializeField] AudioClip uiHover;
    [SerializeField] AudioClip uiSelect;

    void Awake()
    {
        Addressables.LoadAssetAsync<AudioClip>("Assets/Audio/SFX/UIHover.wav")
.ComPLETED += (asyncOp) =>
    { // Attempt to load the hover effect
        if (asyncOp.Status == AsyncOperationStatus.Succeeded)
        { // If the load succeeds
            uiHover = asyncOp.Result; // Assign the hover sound
        }
        else
        {
            Debug.LogError("Failed to load UI audio.");
        }
    };
    Addressables.LoadAssetAsync<AudioClip>("Assets/Audio/SFX/UISelect.wav")
.COMPLETED += (asyncOp) =>
    { // Attempt to load the select effect
        if (asyncOp.Status == AsyncOperationStatus.Succeeded)
        { // If the load succeeds
            uiSelect = asyncOp.Result; // Assign the hover sound
        }
        else
        {
            Debug.LogError("Failed to load UI audio.");
        }
    };
}
```

```

        };
    }

    void Start()
    { // Run OnClick when the button is clicked
        gameObject.GetComponent<Button>().onClick.AddListener(OnClick);
    }

    void OnClick()
    {
        AudioManager.Instance.PlaySFXClip(uiSelect, transform, 1); // Play
uiSelect
    }

    public void OnPointerEnter(PointerEventData eventData)
    { // When the mouse hovers over the button
        AudioManager.Instance.PlaySFXClip(uiHover, transform, 1); // Play
uiHover
    }
}

```

## CASCapsuleVisualiser.cs

```

using UnityEngine;

public class CollideAndSlideCapsuleVisualiser : MonoBehaviour
{
    public Rigidbody player;
    public GameObject playerObj;
    public CapsuleCollider playerCollider;
    public float playerHeight;
    private void OnDrawGizmos()
    { // Draws the top and bottom spheres of the capsule
        Gizmos.DrawWireSphere((player.position + new Vector3(0f,
playerCollider.height * playerHeight / 2f - playerCollider.radius)),
playerCollider.radius);
        Gizmos.DrawWireSphere((player.position - new Vector3(0f,
playerCollider.height * playerHeight / 2f - playerCollider.radius)),
playerCollider.radius);
    }
}

```

## FPS.cs

```

using UnityEngine;
using TMPro;

public class FPS : MonoBehaviour

```

```
{
    public TMP_Text text;
    int number = 0;
    float currentAvg = 0;
    public bool log;

    void Update()
    { // Displays the current framerate, will tank the fps
        float displayVal = 1F / Time.deltaTime;
        text.text = displayVal.ToString();
        if (log)
        {
            Debug.Log(displayVal.ToString());
        }
    }

    float UpdateAverage(float currentFPS)
    { // Optional version that averages the fps across the whole game, not
    very helpful
        number++;
        currentAvg += (currentFPS - currentAvg) / number;
        return currentAvg;
    }
}
```

## IsGroundedTest.cs

```
using UnityEngine;

public class IsGroundedTest : MonoBehaviour
{
    public Rigidbody player;
    public CapsuleCollider playerCollider;
    public GameObject playerObj;
    private void OnDrawGizmos()
    { // Draws the overlap sphere used for depreciated ground check function
        float groundDistance =
player.GetComponent<PlayerMovement>().groundDistance;
        float radius = player.transform.localScale.x * playerCollider.radius;
        float playerHeight = player.transform.localScale.y *
playerCollider.height;
        Debug.DrawLine(player.transform.position, new
Vector3(player.transform.position.x, player.transform.position.y - (0.85f),
player.transform.position.z));
        Gizmos.DrawWireSphere(new Vector3(player.transform.position.x,
player.transform.position.y - groundDistance + radius - playerHeight / 2,
player.transform.position.z), radius);
        // For some reason, in the actual code the distance to the ground has
```

```
[to be multiplied by 2 so that it is 0.85f * 2
    }
}
```

## PosAndVelocity.cs

```
using UnityEngine;
using TMPro;

public class PosAndVelocity : MonoBehaviour
{
    public TMP_Text text;
    public Rigidbody player;

    void Update()
    { // Displays in a text box the player's velocity and position on the 3
    axes
        text.text = $"Pos: x {player.transform.position.x}, y
{player.transform.position.y}, z {player.transform.position.z}<br>Velocity:
{player.linearVelocity}";
    }
}
```

## DialogueInteraction.cs

```
using UnityEngine;
using System.Data;
using System.Collections.Generic;
using System.IO;
using System;
using System.Linq;
using Unity.VisualScripting;
using System.Text;
using Unity.Collections;

public class DialogueInteraction : Interactable
{
    public PlayerManager gameManager;
    private DialogueDatabaseManager ddm;
    public DialogueManager dialogueManager;
    private List<(int, string)> actorsInInteraction = new List<(int,
string)>();
    public UnityEngine.Object dialogueFile;
    private string filepath;
    [SerializeField] private Dictionary<string, int> dialogueSections = new
Dictionary<string, int>();
    private StreamReader fileReader;
```

```

void Start()
{
    gameObject.tag = "Interact"; // Set the object that has this to be
detected as interactable
    gameManager = PlayerManager.Instance; // Fetch the player manager
    ddm = DialogueDatabaseManager.Instance; // Fetch the dialogue database
manager
    dialogueManager = DialogueManager.Instance; // Fetch the dialogue
manager
    filepath = Path.Combine(Application.streamingAssetsPath, "Dialogue",
dialogueFile.name + ".dlg"); // The path to the dialogue file
    PreprocessSections(); // Fetch the dialogue sections
    Debug.Log(ConvertLine("This is a _spooky_ line filled with
<#FFFF00*great* things> ---_and_ *bad* things."));
}
public override void Interact()
{
    (string, string, string[]) output = RetrieveNextLine(); // Get the
next line
    if (output.Item1 == "line")
    { // If it is a line, send it to the dialogue manager
        Debug.Log($"Sending line '{output.Item2 + ":" +
output.Item3[0]}');");
        dialogueManager.DisplayLine(output.Item2 + ":" +
output.Item3[0]);
    }
}
private void PreprocessSections()
{
    StreamReader reader = new StreamReader(filepath);
    int lineNumber = 1; // Start from the beginning of the file
    while (!reader.EndOfStream)
    { // While the end of the file has not been reached
        string line = reader.ReadLine(); // Read the line into a variable
        if (line.StartsWith("[") && line.EndsWith("]"))
        { // If the line starts and ends with square brackets
            dialogueSections.Add(line[1..^1], lineNumber); // Store the
name of the section with the line number
        }
        lineNumber++; // Next line
    }
}
private string RequestNext(bool isDialogue, string language, int lineID)
{
    IDbConnection connection = isDialogue ? ddm.OpenDialogueDb() :
ddm.OpenActorDb(); // If the database needed is dialogue, open that table,
otherwise open the actor table
    IDbCommand lineRequest = connection.CreateCommand(); // Begin a query
}

```

```

        lineRequest.CommandText = $"SELECT {language} FROM {(isDialogue ?
"Dialogue" : "Actor")}" WHERE id={lineID}"; // Select the language of ID lineID
from the right table
        IDataReader response = lineRequest.ExecuteReader(); // Execute the
query
        bool success = response.Read(); // Read a line and store it
        if (!success)
        { // If the query failed
            Debug.LogError($"{{(isDialogue ? "Line" : "Actor")}} {lineID} is not
in {{(isDialogue ? "Dialogue" : "Actor")}} table");
            connection.Close();
            return ""; // Return an empty string
        }
        string line = response.GetString(0); // Get the first string
        connection.Close(); // Close the database
        return line; // Return the line
    }
    private string ConvertLine(string line)
    {
        string copy = "";
        bool firstItalics = true; // Tracks whether the first underscore is
yet to come
        bool firstBold = true; // Tracks whether the first asterisk is yet to
come
        for (int i = 0; i < line.Length; i++)
        { // For every character in the line
            if (line[i] == '\\')
            { // Ignore the next character
                i++; // Skip over the next character
            }
            else if (line[i] == '<')
            { // Begin reading a colour
                string hexColour = line.Substring(i + 7); // Grab a colour
                code
                copy += "<color=" + hexColour + ">"; // Add the hexcode along
                with the <color= >
                i += 6; // Skip over the hexcode
            }
            else if (line[i] == '>')
            {
                copy += "</color>"; // Replace with the ending tag
            }
            else if (line[i] == '_')
            {
                if (firstItalics)
                {
                    copy += "<i>"; // Replace with the starting italics tag
                    firstItalics = false;
                }
            }
        }
    }
}

```

```

        }
        else
        {
            copy += "</i>"; // Replace with the ending italics tag
            firstItalics = true;
        }
    }
    else if (line[i] == '*')
    {
        if (firstBold)
        {
            copy += "<b>"; // Replace with the starting bold tag
            firstBold = false;
        }
        else
        {
            copy += "</b>"; // Replace with then ending bold tag
            firstBold = true;
        }
    }
    else
    {
        copy += line[i]; // Just copy the character
    }
}
return copy;
}
// DLG Interpreter
private int lineNumber = 2; // The line of the file that the interpreter
is on
/*
-> is the goto operator
[] is dialogue section
#Ax is Actor database
#Lx is Line database
:: actor line/actor split
@ is variable
= is variable assign
*/
public (string, string, string[]) RetrieveNextLine()
{
    while (true)
    {
        TreeNode tokenTree = Tokenizer(); // Tokenize the line
        (string, string, string[]) returnVals = Analyzer(tokenTree); // Analyse the tokens of the lines
        string type = returnVals.Item1;
        if (type == "true")

```

```

        { // If the analyzer has decided another line must be read
            continue; // Read another line
        }

        if (type == "line")
        {
            string actor = returnVals.Item2; // Retrieve the actor
            string line = ConvertLine(returnVals.Item3[0]); // Convert the
            lone line
            string[] lines = new string[1]; // Make a new array of length
            1 to store the converted line
            lines[0] = line; // Store the line in the return output
            return (type, actor, lines); // Return as appropriate
        }
        else if (type == "choice")
        { // If the node is a choice node
            string[] lines = returnVals.Item3; // Fetch all the dialogue
            options
            string[] convertedLines = new string[lines.Length]; // Make a
            new array for however many dialogue options
            for (int i = 0; i < lines.Length; i++)
            { // For each option
                convertedLines[i] = ConvertLine(lines[i]); // Convert it
                and assign to the return output
            }
            return (type, returnVals.Item2, convertedLines); //
            returnVals.Item2 will be blank as the actor is assumed to be the player, but
            this must be passed through to fulfill the return type conditions
        }
        /*else if (type == "facing")
        {

        } */
        else
        {
            throw new Exception($"Line Retrieval: '{type}' was not a
            recognised type.");
        }
    }
    private TreeNode Tokenizer()
    {
        TreeNode fileTree = new TreeNode(); // This is here in case a choice
        exists; in which case multiple lines need to be read at once
        using (StreamReader fileReader = new StreamReader(filepath)){
            for (int i = 0; i < lineNumber - 1; i++)
            {
                fileReader.ReadLine(); // Skip lines until the current one is

```

```

reached
    }
    int lastLineNumber = lineNumber - 1;
    string line = fileReader.ReadLine(); // Read the current line
    while (!fileReader.EndOfStream && lastLineNumber != lineNumber)
    {
        if (line.Length < 3)
            { // The minimum length of the line cannot go below 3
                throw new Exception($"Tokenizer: The line '{line}' is
missing information.");
            }
        byte[] lineBytes = Encoding.UTF8.GetBytes(line); // Convert
into a list of characters
        MemoryStream stream = new MemoryStream(lineBytes); // Make
into a stream for StreamReader later
        lastLineNumber = lineNumber; // Set the previous line
        StreamReader lineReader = new StreamReader(stream); // Prepare
for reading per character
        TreeNode lineTree = new TreeNode(); // New node for the line
        lineTree.AddType("linetree");
        fileTree.AddChild(lineTree); // Add the node to the file tree
        ReadNext(lineReader, lineTree); // Tokenize the line
        bool shouldStay = false; // Initialise shouldStay
        if (fileTree.GetChildren()[^1].GetChildren()[^1].GetNodeType() ==
"choice")
            { // If the node is a choice node
                if
(fileTree.GetChildren()[^1].GetChildren()[^1].value["end"] == "false")
                    { // If it is not the last node in the choice - it is
necessary to put this on a separate line to avoid an error
                        shouldStay = true; // Require reading the next line
                    }
            }
        string nextLine = fileReader.ReadLine(); // Check the next
line

        if (nextLine[0] != '[' && nextLine != null)
            { // If the next line isn't a dialogue section header or the
end of the file
                lineNumber++; // Now on the next line
                line = nextLine; // Set the current line
                if (!shouldStay)
                    { // If staying is not required
                        break; // Exit
                    } else {
                        throw new Exception("Expected another line to complete
the choice: maybe you have an extra comma?");
                    }
            }
    }
}

```

```

        }
    }
}
return fileTree; // Return the read lines
}
// Tokenizer
private void ReadNext(StreamReader line, TreeNode lineTree)
{
    while (!line.EndOfStream)
    { // While the end of the line hasn't been reached
        char token = ReadChar(line);
        if (token == '#')
        {
            ReadID(line, lineTree);
        }
        else if (token == '@')
        {
            ReadIf(line, lineTree);
        }
        else if (token == '-')
        {
            ReadJump(line, lineTree);
        }
        else if (token == '|')
        {
            ReadChoice(line, lineTree);
        }
        else if (token == ':')
        {
            token = ReadChar(line);
            if (token == ':')
            {
                ReadSeparator(line, lineTree);
            }
            else
            {
                throw new Exception($"Tokenizer: The starting character '{token}' was not recognised.");
            }
        }
    }
} /*else if (token == '-'){
    ReadAnimation(line, lineTree);
}*/
private char ReadChar(StreamReader line)
{
    int character = 32; // ASCII for space
}

```

```
        while (character == 32)
        { // Skips over spaces
            character = line.Read();
        }
        return character == -1 ? throw new Exception("Tokenizer: Nothing left
to read.") : (char)character; // If it is not the end of the line, return the
char
    }
    private char PeekChar(StreamReader line)
    {
        int character = 32; // ASCII for space
        while (character == 32)
        { // Skips over spaces
            character = line.Peek();
        }
        return character == -1 ? throw new Exception("Tokenizer: Nothing left
to peek.") : (char)character; // If it is not the end of the line, return the
char
    }
    private void ReadID(StreamReader line, TreeNode lineTree)
    {
        char character = ReadChar(line); // Read the next character
        int id = ReadNumber(line); // Read the number attached to it
        TreeNode idNode = new TreeNode(); // Make a new ID node
        lineTree.AddChild(idNode); // Add it to the line's children

        if (character == 'A')
        { // If the ID is an actor
            idNode.AddType("actor");
        }
        else if (character == 'L')
        { // If the ID is a line
            idNode.AddType("line");
        }
        else
        {
            throw new Exception($"Tokenizer: Unable to decide if this is an
actor or line reference: {character}");
        }
        idNode.value.Add("id", id.ToString()); // Add the number to the node's
dictionary
    }
    private int ReadNumber(StreamReader line)
    {
        string num = "";
        while (!line.EndOfStream)
        { // While the end of the line hasn't been reached
            char character = PeekChar(line); // Peek a character
```

```

        if (int.TryParse(character.ToString(), out _))
            { // If it is a number
                num += ReadChar(line); // Read it and add it to the number
            }
        else
        {
            break;
        }
    }
    return int.Parse(num); // Return the number
}
private void ReadIf(StreamReader line, TreeNode lineTree)
{
    TreeNode ifNode = new TreeNode(); // Make a new node
    lineTree.AddChild(ifNode); // Add it to lineTree's children
    ifNode.AddType("if"); // Set the type to 'if'
    ReadVariable(line, ifNode); // Read the variable and add it to
    ifNode's children
    ReadOperator(line, ifNode); // Read the operator and add it to
    ifNode's children
    char character = PeekChar(line); // Peek at the next character
    if (char.IsNumber(character))
    { // If it is a number
        TreeNode numNode = new TreeNode();
        ifNode.AddChild(numNode); // Make a new child node to store the
        number
        numNode.AddType("num"); // Set the type to 'num'
        numNode.value.Add("num", ReadNumber(line).ToString()); // Assign
        the number to numNode
    }
    else if (char.IsLetter(character) || character == '_')
    { // If the next character is a letter or _
        ReadVariable(line, ifNode); // Read a variable
    }
    else
    {
        throw new Exception("Tokenizer: Nothing valid to compare to.");
    }
}
private void ReadVariable(StreamReader line, TreeNode lineTree)
{
    char character = PeekChar(line); // Check what the next character is
    if (char.IsNumber(character))
    { // A variable name can't start with a number
        throw new Exception("Tokenizer: Variable can't start with a
        number!");
    }
}

```

```

TreeNode variableNode = new TreeNode(); // Make a new node
lineTree.AddChild(variableNode); // Add it to lineTree's children
string variable = ""; // Define variable
while (!line.EndOfStream)
{ // Until the end of the line
    character = PeekChar(line); // Peek at the next character
    if (char.IsLetterOrDigit(character) || character == '_')
    { // If the character is valid
        variable += ReadChar(line); // Read it and add it to the name
    }
    else
    {
        break;
    }
}
variableNode.AddType("var"); // Add the type 'var'
variableNode.value.Add("var", variable); // Add the name of the
variable
}
private string ReadOperator(StreamReader line, TreeNode lineTree)
{
    string op = ""; // Create a variable to hold the operator
    char[] opChars = { '!', '>', '<', '=' }; // List of the acceptable
    characters
    TreeNode opNode = new TreeNode(); // Make a new node
    lineTree.AddChild(opNode); // Add it to lineTree's children
    while (!line.EndOfStream)
    { // While not the end of the line
        char character = PeekChar(line); // Check the next character
        if (op.Length == 0 && opChars.Contains(character))
        { // If nothing has been read and it's a valid character
            op += ReadChar(line); // Add it to the string
        }
        else if (op.Length == 1 && opChars.Contains(character))
        { // If it is a comparison
            op += ReadChar(line); // Add the character to the string
            opNode.AddType("comp"); // Assign the type as comparison
            opNode.value.Add("op", op); // Add the comparison type
            return op;
        }
        else if (op.Length == 1 && op == "=")
        { // If it is only an assignment
            opNode.AddType("assign"); // Set the type as assignment
            return op;
        }
        else
        {
            throw new Exception($"Tokenizer: Invalid operator '{op}'.");
        }
    }
}

```

```
        }
    }
    throw new Exception($"Tokenizer: No valid operator found: '{op}'");
}
private void ReadJump(StreamReader line, TreeNode lineTree)
{
    char character = ReadChar(line); // Read the next character
    if (character == '>')
    { // If it is an arrow
        string section = ""; // Define a variable for the section name
        while (!line.EndOfStream)
        { // Copy all other characters as the section name
            section += ReadChar(line);
        }
        TreeNode sectionNode = new TreeNode(); // Create a new node
        lineTree.AddChild(sectionNode); // Add it to lineTree's children
        sectionNode.AddType("jump"); // Set the type as jump
        sectionNode.value.Add("section", section); // Set the section name
    }
    else
    {
        throw new Exception("Tokenizer: This is not a jump, what were you
trying to signify?");
    }
}
private void ReadChoice(StreamReader line, TreeNode lineTree)
{
    TreeNode choiceNode = new TreeNode(); // Make a new node
    choiceNode.AddType("choice"); // Set the type as choice
    lineTree.AddChild(choiceNode); // Add to lineTree's children
    while (!line.EndOfStream)
    { // If it isn't the end of the line
        char character = ReadChar(line); // Get the next character

        if (character == '#')
        { // Read an ID
            ReadID(line, lineTree);
        }
        else if (character == ',')
        { // If there is a comma
            choiceNode.value.Add("end", "false"); // Mark that there will
be another line
            return;
        }
        else
        {
            throw new Exception($"Tokenizer: Unrecognised character
'{character}' following choice line");
        }
    }
}
```

```

        }
    choiceNode.value.Add("end", "true"); // No comma = last option
}
private void ReadSeparator(StreamReader line, TreeNode lineTree)
{ // Required to check for proper syntax
    TreeNode separatorNode = new TreeNode(); // Make a new node
    separatorNode.AddType("sep"); // Set the type as a separator
    lineTree.AddChild(separatorNode); // Add to lineTree's children
}/*
private void ReadAnimation(StreamReader line, TreeNode lineTree){
    char character = PeekChar(line);
    if (character == '-'){
        ReadChar(line);
        TreeNode animNode = new TreeNode();
        lineTree.AddChild(animNode);
        animNode.AddType("anim");
        animNode.value.Add("num", ReadNumber(line).ToString());
    }
}/*
// Analysis
private (string, string, string[]) Analyzer(TreeNode fileTree)
{
    string[] lines = new string[fileTree.GetChildren().Count]; // Make a
new return array based on the number of lines
    string returnType = "none"; // Default return type
    string[] returnLines = new string[lines.Length]; // Copy of lines
    for (int i = 0; i < fileTree.GetChildren().Count; i++)
    { // For each line
        TreeNode line = fileTree.GetChildren()[i]; // Get the line
        List<TreeNode> elements = line.GetChildren(); // Get the elements
of the line
        if (elements[0].GetNodeType() == "actor")
        { // If the line is an actor
            /*
            if (elements[1].GetNodeType() == "anim")
            {
                returnType = "animation";
                string returnActor = RequestNext(false, "actor",
Int32.Parse(elements[0].value["id"]));
                string[] animID = { elements[1].value["num"] };
                return (returnType, returnActor, animID);
            } else if (elements[1].GetNodeType() == "sep" &&
elements[2].GetNodeType() == "actor"){
                returnType = "facing";
                string returnActor1 = RequestNext(false, "actor",
Int32.Parse(elements[0].value["id"]));
                string[] returnActor2 = {RequestNext(false, "actor",

```

```

        Int32.Parse(elements[2].value["id"])));
            returnType = "returnType, returnActor1, returnActor2);
        }
        else */
        /* if (elements[1].GetNodeType() == "anim")
        {
            returnType = "anim";
            string returnActor = RequestNext(false, "actor",
Int32.Parse(elements[0].value["id"]));
            string[] animID = {elements[1].value["num"]};
            return (returnType, returnActor, returnLines);
        } else */
        if (elements[1].GetNodeType() == "sep" &&
elements[2].GetNodeType() == "line")
        { // If a separator and a line node follow the actor
            returnType = "line"; // Set the return type to line
            returnLines[i] = LineAnalysis(elements.GetRange(2,
elements.Count - 2)); // Analyse all the lines into 1
            string returnActor = RequestNext(false, "actor",
Int32.Parse(elements[0].value["id"])); // Get the actor from the database
            return (returnType, returnActor, returnLines); // Return
the values
        }
        else
        {
            foreach (TreeNode element in elements){
                Debug.Log(element.GetNodeType());
            }
            throw new Exception("Analysis: Unable to progress after
discovering actor in analysis.");
        }
    }
    else if (elements[0].GetNodeType() == "jump")
    {
        returnType = "true"; // Tell RequestNextLine that it needs to
jump to another line
        JumpToSection(elements[0]); // Get the line number of the
section
        return (returnType, null, null);
    }
    else if (elements[0].GetNodeType() == "if")
    {
        returnType = "false";
        int var1 = AnalyseIf(elements[2]); // Analyse the third
element
        if (elements[1].GetNodeType() == "assign")
        { // If the element is getting assigned
            if (elements[2].value["var"] == "choice")

```

```

        {
            // If the third element was the variable choice
            choice = var1; // Set choice to the value of var1
        }
        else
        {
            gameManager.SetVarValue(elements[2].value["var"],
var1); // Tell the game manager to assign the variable in the save data
        }
    }
    else if (elements[1].GetNodeType() == "comp")
    { // If the variable is being compared
        int var2 = AnalyseIf(elements[3]); // Analyse the next
variable
        bool passed = EvalComparison(var1,
elements[1].value["op"], var2); // Compare the variables
        if (passed)
        { // If the comparison was true
            if (elements[4].GetNodeType() == "jump")
            { // If the next node is a jump node
                JumpToSection(elements[4]); // Set the line number
to the section mentioned
                returnType = "true"; // Tell RequestNextLine that
it needs to jump to another line
            }
            else
            {
                throw new Exception("Analysis: Expected a 'jump'
node with a section.");
            }
        }
        return (returnType, null, null);
    }
    else if (elements[0].GetNodeType() == "choice")
    {
        returnType = "choice";
        TreeNode child = elements[0].GetChildren()[0];
        returnLines[i] = LineAnalysis(elements.GetRange(1,
elements.Count - 1));
        // Not fully implemented
    }
    else
    {
        throw new Exception($"Analysis: Unrecognised node type
'{elements[0].GetNodeType()}' found in syntax analysis step.");
    }
}
return (returnType, null, returnLines);

```

```

    }

    private int choice = 0;
    private int AnalyseIf(TreeNode element)
    {
        if (element.GetNodeType() == "var")
        { // If the node is variable
            if (element.value["var"] == "choice")
            { // If that variable is choice
                return choice; // Return the value of choice
            }
            // Get value of variable
            return gameManager.GetVarValue(element.value["var"]); // Request
the game manager for the value of the variable from the save data
        }
        else if (element.GetNodeType() == "num")
        { // If the element was a number
            return int.Parse(element.value["num"]); // Cast the number
        }
        else
        {
            throw new Exception($"Analysis: 'if' came across node of type
{element.GetNodeType()} instead of a var or num");
        }
    }
    private string LineAnalysis(List<TreeNode> lines)
    {
        string returnLine = ""; // Store the output line
        for (int j = 0; j < lines.Count; j++)
        { // For every dialogue line
            returnLine += RequestNext(true, "english",
Int32.Parse(lines[j].value["id"])) + " "; // Request it from the database then
append it the output
        }
        returnLine = returnLine[..^1]; // Remove the trailing space
        return returnLine;
    }
    private void JumpToSection(TreeNode jumpNode)
    {
        string section = jumpNode.value["section"]; // Get the line number of
the section
        lineNumber = dialogueSections[section] + 1; // Set the new line number
    }
    private bool EvalComparison(int variable, string op, int compareTo)
    {
        switch (op)
        { // Compare based on the operator
            case "==":
                return variable == compareTo;
        }
    }
}

```

```

        case ">=":
            return variable >= compareTo;
        case "<=":
            return variable <= compareTo;
        case "!=":
            return variable != compareTo;
        default:
            throw new Exception($"Analysis: {op} is not a valid comparison
operator");
    }
}
}

```

## TreeNode.cs

```

using System.Collections.Generic;
using UnityEngine;

public class TreeNode
{
    public Dictionary<string, string> value = new Dictionary<string,
string>(); // Dictionary for values
    private List<TreeNode> children = new List<TreeNode>(); // List of
children
    public void AddChild(TreeNode child)
    { // Add a child
        children.Add(child);
    }
    public List<TreeNode> GetChildren()
    { // Get every child
        return children;
    }
    public void AddType(string typeValue)
    { // Add a type to the node
        Debug.Log($"Adding type '{typeValue}'");
        value.Add("type", typeValue); // Add the type
    }
    public string GetNodeType()
    { // Get the node's type
        return value["type"];
    }
}

```

## Entity.cs

```

using UnityEngine;

```

```
public abstract class Entity : MonoBehaviour
{
    PlayerManager gameManager;
    [SerializeField] private float maxHealth = 100;
    [SerializeField] private float health = 100;
    [SerializeField] private float defenseMultiplier = 0;
    [SerializeField] private float healingMultiplier = 0;
    public float MaxHealth { get { return maxHealth; } set { maxHealth = value; } }
    public float Health { get { return health; } set { health = value; } }
    public float DefenseMultiplier { get { return defenseMultiplier; } set { defenseMultiplier = value; } }
    public float HealingMultiplier { get { return healingMultiplier; } set { healingMultiplier = value; } }
    public virtual int merit => 1;

    public virtual void Start()
    {
        Health = MaxHealth; // Sets the health to the maximum health
        gameManager = PlayerManager.Instance; // Fetches a reference to the game manager
    }

    public void Damage(float attack)
    {
        Debug.Log("Ouch " + attack);
        Health -= attack * (1 - DefenseMultiplier); // Reduce the damage taken by 1 - defenseMultiplier
        Health = Mathf.Clamp(Health, 0, MaxHealth); // Make sure health cannot go below zero or above maximum health
        if (Health <= 0)
        { // Only kills when it takes damage
            Kill(); // Kill the object
        }
    }

    public void Healing(float heal)
    {
        Health += heal * (1 + HealingMultiplier); // Increases healing based on the healing multiplier
        Health = Mathf.Clamp(Health, 0, MaxHealth); // Makes sure that the entity does not heal above what it can handle
    }

    public virtual void Kill()
    {
        Debug.Log($"Kill requested on {gameObject}");
        gameManager.AddMerit(merit); // Award merit on death
        Destroy(gameObject); // Destroy self
    }
}
```

## PlayerEntity.cs

```
using TMPro;
using UnityEngine;
using UnityEngine.AddressableAssets;
using UnityEngine.ResourceManagement.AsyncOperations;
using UnityEngine.UI;

public class PlayerEntity : Entity
{
    public Slider slider;
    public TMP_Text healthText;
    bool run = true;

    [SerializeField] private AudioClip death;

    void Awake()
    {
        Addressables.LoadAssetAsync<AudioClip>("Assets/Audio/SFX/PlayerDeath.wav").Completed += (asyncOp) =>
        { // Load the death sound effect
            if (asyncOp.Status == AsyncOperationStatus.Succeeded)
            { // If succeeded
                death = asyncOp.Result; // Assign the death sound
            }
            else
            {
                Debug.LogError("Failed to load player death audio.");
            }
        };
    }

    public override void Start()
    {
        Health = MaxHealth;
        if (slider == null)
        {
            Debug.Log("Health slider not assigned");
            run = false;
        }
        else if (healthText == null)
        {
            Debug.Log("Health text not assigned.");
            run = false;
        }
    }
}
```

```

    }

    public void UpgradeHealth(float extra)
    {
        MaxHealth += extra; // Add onto the maximum health the added amount
    }

    private void Update()
    {
        if (run)
        {
            slider.value = Health / MaxHealth; // Set the slider value to be
            the fraction of the health out of the maximum health
            healthText.text = Health.ToString(); // Set the health text to the
            current remaining health
        }
    }

    public override void Kill()
    {
        PlayerManager.Instance.inputs.Player.Disable(); // Disable the Player
        input map
        PlayerManager.Instance.PlayerDeath(); // Tell the manager that the
        player has died
        MenuManager.Instance.ChangeMenu("Death"); // Change to the death menu
        AudioManager.Instance.PlaySFXClip(death, transform, 1f); // Play a
        death sound effect
    }
}

```

## EnemyEntity.cs

```

using UnityEngine;

public class EnemyEntity : Entity
{
    public EnemyManager manager;
    public GameObject target;
    public Vector3 searchingTarget;
    [SerializeField] protected bool searchTargetSet = false;
    public EnemyManager.MemberState state;
    protected Rigidbody rb;
    public float maxMoveSpeed = 100;
    public virtual float MaxMoveSpeed => maxMoveSpeed;
    public virtual float RotationSpeed => 10f;
    public virtual float FarTargetRange => 15f;
    public virtual float NearTargetRange => 2.5f;
    protected float midRange;
}

```

```

public virtual float DetectionRange => 20f;
protected int rayCount;
protected float rayAngle;
protected float maxSlope;
// should probably also have detection angle as well for long distance
protected LayerMask playerLayer;
//protected Vector3 groundNormal = Vector3.up;

[SerializeField] protected GameObject weapon;
protected Weapon weaponScript;

[SerializeField] protected Vector3 velocity;

public override int merit => 10;

[SerializeField] protected float proportionalGain = 1;
[SerializeField] protected float derivativeGain = 0.1f;
[SerializeField] protected float integralGain = 2;
protected Vector3 lastError = Vector3.zero;
protected Vector3 lastPosition = Vector3.zero;
[SerializeField] protected Vector3 storedIntegral = Vector3.one;
[SerializeField] protected float maxStoredIntegral = 2;
public enum DerivativeType
{
    Velocity,
    Error,
}
[SerializeField] public DerivativeType derivativeType =
DerivativeType.Error;
protected bool initialised = false;

protected virtual void Awake()
{
    SetAngles(); // work out the number of rays and the angle between them
    //AssignConsts(); // Fetch constats
    midRange = (FarTargetRange + NearTargetRange) / 2; // Calculate the
position at which the enemy should be at
}

public void SetAngles()
{ // Finds the angle needed to make sure that the player is not missed
between rays
    float quarterCircumference = 2 * Mathf.PI * DetectionRange * 0.25f; //
Find the arc length
    rayCount = Mathf.CeilToInt(quarterCircumference); // Calculate the
number of rays
    rayAngle = 90 / quarterCircumference; // Calculate the angle between
each ray
}

```

```
        }

    public override void Start()
    {
        Health = MaxHealth;
        rb = GetComponent<Rigidbody>();
        playerLayer = LayerMask.GetMask("Player");
    }

    protected virtual void FixedUpdate()
    {
        EnemyLoop();
    }

    protected virtual void EnemyLoop()
    {
        /* search for enemies and move around
        If enemies found switch to combat and alert manager
        */
        velocity = rb.linearVelocity; // Store a copy of the current velocity
        for debugging
        Debug.DrawRay(rb.transform.position, rb.transform.rotation *
        Vector3.forward, Color.yellow);
        if (state != EnemyManager.MemberState.None)
        {
            // Required for the first frame when no target is there or other
            cases
            if (target != null)
            {
                RotateToTarget(target.transform); // Rotate towards the target
            }
            if (state == EnemyManager.MemberState.Combat)
            {
                // Move towards and fire
                Vector3 movementTarget = target.transform.position -
                (target.transform.position - rb.transform.position).normalized * midRange;
                ApplyMovement(movementTarget); // Calculate the new movement
                to the target
                Combat(); // Check if the weapon can be fired
            }
            else if (state == EnemyManager.MemberState.Searching)
            {
                // Search target is reached if they are within 1 metre of it
                if ((searchingTarget - rb.transform.position).magnitude < 1 ||
                !searchTargetSet)
                {
                    SelectTarget(); // Select the next searching target
                    ResetInitialisation(); // Skip calculating the derivative
                    for this frame
                    searchTargetSet = true; // Skips a potential problem on
                    the first frame
                }
            }
        }
    }
}
```

```

        }
        ApplyMovement(searchingTarget); // Calculate the new movement
to the target
        Searching(); // Search at the new position
    }
}
lastPosition = rb.transform.position; // Store a copy of the current
position
}
/*
To improve the system, it should check whether the player is still visible
to the enemy each time, and it should also simultaneously update the searching
target.
This will be used as the backup when the player is out of view, and will
be used as the last known location of the player
Also start a timer which will have range of how long it is (random time
within) which determines when the enemy loses aggro
Once the player has been seen and the manager has been updated, raise the
awareness of the enemies in that group - might search more aggressively or
detect the player faster
Could implement a sound detection system with priority levels
*/
protected virtual void Combat()
{ // Fire weapon
    RaycastHit hitInfo;
    if (Physics.Raycast( // Check if there the player is in the weapon's
view
        origin: rb.transform.position,
        direction: weapon.transform.GetChild(0).forward,
        hitInfo: out hitInfo,
        maxDistance: DetectionRange,
        layerMask: ~LayerMask.GetMask("Weapon")
    ))
    {
        GameObject hit = hitInfo.collider.gameObject; // Grab the hit
gameobject
        Debug.Log($"Enemy {gameObject.GetInstanceID()} ({gameObject.name})"
facing {hit.GetInstanceID()} ({hit.gameObject.name})");
        Entity entityClass = hit.GetComponent<Entity>(); // Get the entity
class from the object
        if (entityClass != null)
            { // If the object is damageable
                weaponScript.Fire(); // Fire the weapon
                Debug.Log($"Enemy {gameObject.GetInstanceID()}"
({gameObject.name}) fired its {weaponScript.WeaponType}");
            }
    }
}

```

```

// At some point absolutely replace this with the NavMesh
protected virtual void ApplyMovement(Vector3 target)
{ // Fetch changes in movement
    float movementX = PIDUpdate(Time.fixedDeltaTime,
rb.transform.position.x, target.x, ref lastPosition.x, ref lastError.x, ref
storedIntegral.x);
    float movementY = 0;
    float movementZ = PIDUpdate(Time.fixedDeltaTime,
rb.transform.position.z, target.z, ref lastPosition.z, ref lastError.z, ref
storedIntegral.z);
    Vector3 movementTotal = new Vector3(movementX, movementY, movementZ);
// Combine the movement
    if (movementTotal.magnitude > MaxMoveSpeed)
    { // Limit the movement
        movementTotal *= MaxMoveSpeed / movementTotal.magnitude; // set
movementTotal's magnitude to the maxMoveSpeed
    }
    Debug.DrawRay(rb.transform.position, new Vector3(movementX, 0f,
0f).normalized, Color.red); // X direction
    Debug.DrawRay(rb.transform.position, new Vector3(0f, movementY,
0f).normalized, Color.green); // Y direction
    Debug.DrawRay(rb.transform.position, new Vector3(0f, 0f,
movementZ).normalized, Color.blue); // Z direction
    Debug.DrawRay(rb.transform.position, movementTotal.normalized,
Color.black); // Total direction
    rb.AddForce(movementTotal, ForceMode.VelocityChange); // Apply
movement
}
//TODO Change to an overlap sphere checking angles instead and a singular
downwards raycast with no max distance
// Allows for changing of the looking direction
protected void Searching()
{
    for (int i = 0; i < rayCount; i++)
    {
        RaycastHit hitInfo;
        Vector3 rayDirection = Quaternion.AngleAxis(-45f + rayAngle * i,
Vector3.up) * Vector3.forward; // Calculate ray direction
        rayDirection = rb.transform.rotation * rayDirection; // Make it
relative to the entity
        Debug.DrawRay(rb.transform.position, rayDirection, Color.red);
        if (Physics.Raycast( // Check if a ray has intersected with a
target
            origin: rb.transform.position,
            direction: rayDirection,
            hitInfo: out hitInfo,
            maxDistance: DetectionRange,
            layerMask: playerLayer
}

```

```
        ))
    {
        Debug.DrawRay(rb.transform.position, rayDirection,
Color.green);
        state = EnemyManager.MemberState.Combat; // Set state to
combat
        target = hitInfo.collider.gameObject; // Set target to the
player's gameobject
        break;
    }
}
//UpdateManager(EnemyManager.MemberState.Combat);
protected void SelectTarget()
{
    bool found;
    Vector3 point = GetClosestPoint(out found);
    if (!found)
    { // If piece of level geometry is not found within the entity's
detection range
        searchingTarget.y = rb.transform.position.y - DetectionRange; // /
Set the target to be below it.
        return;
    }
    Vector3 entityToPoint = rb.transform.position - point; // calculate
the distance to the found point
    if (entityToPoint.magnitude > DetectionRange)
    { // If the point is outside the range of the entity
        searchingTarget = point - entityToPoint.normalized *
Random.Range(0, DetectionRange); // Set the target distance as a random
distance towards the closest point.
        return;
    }

    bool accessible = false;
    while (!accessible)
    { // While the target is not accessible, generate a new point
        searchingTarget.x = Random.Range(-DetectionRange, DetectionRange);
// Generate X point
        searchingTarget.y = Random.Range(-DetectionRange, DetectionRange);
// Generate Y point
        searchingTarget.z = Random.Range(-DetectionRange, DetectionRange);
// Generate Z point
        searchingTarget = searchingTarget + rb.transform.position; // Make
the point relative to the current position of the player
        RaycastHit hitInfo;
        if (Physics.Raycast( // Check that there is no object between the
target and the entity
```

```

        origin: rb.transform.position,
        direction: (searchingTarget -
rb.transform.position).normalized,
        hitInfo: out hitInfo,
        maxDistance: DetectionRange,
        layerMask: playerLayer
    ))
{
    if ((hitInfo.point - rb.transform.position).magnitude < 1f)
    { // If the point is too close to the entity
        continue;
    }
    else
    { // The point is set
        searchingTarget = hitInfo.point;
        accessible = true; // Used to exit the loop
    }
}
ResetInitialisation(); // Reset the derivative
}
protected Vector3 GetClosestPoint(out bool found)
{
    Collider[] colliders = Physics.OverlapSphere(
        position: rb.transform.position,
        radius: DetectionRange * 2
    ); // Get list of colliders nearby
    found = false;
    Vector3 point = Vector3.zero;
    float distSqrD = Mathf.Infinity; // Start off with an infinite
distance
    foreach (Collider collider in colliders)
    {
        found = true;
        if ((rb.transform.position -
collider.transform.position).sqrMagnitude < distSqrD)
            { // if the distance to this collider is less than the current
smallest
                distSqrD = (rb.transform.position -
collider.transform.position).sqrMagnitude; // Set its distance as the new
smallest one
                point = collider.ClosestPoint(rb.transform.position); // Get
the exact distance to the point
            }
    }
    return point;
}
private void UpdateManager(EnemyManager.MemberState state)

```

```

    {
        // send new state to manager
    }

    protected virtual float PIDUpdate(float timeStep, float currentValue,
float targetValue, ref float lastPosition, ref float lastError, ref float
storedIntegral)
    {
        float error = targetValue - currentValue; // Calculate the difference
        float force = proportionalGain * error; // Calculate the P term
        float derivative = 0; // Default derivative value
        if (initialised)
        { // If the target has not recently changed
            if (derivativeType == DerivativeType.Error)
            { // If using the error to calculate the derivative
                if (error - lastError != 0)
                { // Verify that an error will not occur
                    derivative = (error - lastError) / timeStep; // Rate of
change
                }
            }
            else
            { // If using the position to calculate the derivative
                if (currentValue - lastPosition != 0)
                { // Verify that an error will not occur
                    derivative = (currentValue - lastPosition) / timeStep; // Rate of change
                }
            }
        }
        else
        { // If the target has recently changed
            initialised = true; // Allows skipping of the error
        }
        derivative *= derivativeGain;
        storedIntegral = Mathf.Clamp(storedIntegral + error * timeStep, -
maxStoredIntegral, maxStoredIntegral); // Make sure the stored errors do not
exceed the limit
        float integral = integralGain * storedIntegral; // Calculate the I
term

        lastError = error; // Store the current error value for the next
iteration
        return force + integral + derivative; // Return the total
    }

    public void ResetInitialisation() => initialised = false;
    protected virtual void RotateToTarget(Transform target)
}

```

```

    {
        Vector3 targetDirection = target.transform.position -
weapon.transform.position; // Get the direction from the weapon to the target
        targetDirection.y = 0;
        Quaternion dirInQuaternion =
Quaternion.LookRotation(targetDirection.normalized, Vector3.up); // Rotation
for the next step
        rb.transform.rotation = Quaternion.Slerp(rb.transform.rotation,
dirInQuaternion, RotationSpeed * Time.fixedDeltaTime); // Rotate the enemy so
that the gun is in position to fire the player
    }
    public override void Kill()
    {
        weapon.GetComponent<Rigidbody>().useGravity = true; // Allow the
weapon to react normally
        weapon.GetComponent<Rigidbody>().detectCollisions = true; // Allow it
to detect collision
        weapon.GetComponent<Rigidbody>().isKinematic = true; // Allow it to be
moved by the Physics system
        weapon.transform.SetParent(null); // Release the weapon
        PlayerManager.Instance.AddMerit(merit); // Award merit on death
        Destroy(gameObject); // Destroy self
    }
}

```

## Drone.cs

```

using UnityEngine;

public class Drone : EnemyEntity
{
    public override float MaxMoveSpeed => maxMoveSpeed;
    public override float FarTargetRange => 25f;
    public override float NearTargetRange => 5f;
    public override float DetectionRange => 30f;
    public override int merit => 15;

    public override void Start()
    {
        Health = MaxHealth;
        rb = GetComponent<Rigidbody>();
        playerLayer = LayerMask.GetMask("Player");
    }
    protected override void FixedUpdate()
    {
        EnemyLoop();
    }
    protected override void EnemyLoop()

```

```

{
    /* search for enemies and move around
    If enemies found switch to combat and alert manager
    */
    velocity = rb.linearVelocity; // Store a copy of the current velocity
for debugging
    Debug.DrawRay(rb.transform.position, rb.transform.rotation *
Vector3.forward, Color.yellow);
    if (state != EnemyManager.MemberState.None)
    {
        // Required for the first frame when no target is there or other
cases
        if (target != null)
        {
            RotateToTarget(target.transform); // Rotate towards the target
        }
        if (state == EnemyManager.MemberState.Combat)
        {
            // Move towards and fire
            Vector3 movementTarget = target.transform.position -
(target.transform.position - rb.transform.position).normalized * midRange;
            ApplyMovement(movementTarget); // Calculate the new movement
to the target
            Combat(); // Check if the weapon can be fired
        }
        else if (state == EnemyManager.MemberState.Searching)
        {
            // Search target is reached if they are within 1 metre of it
            if ((searchingTarget - rb.transform.position).magnitude < 1 ||
!searchTargetSet)
            {
                SelectTarget(); // Select the next searching target
                ResetInitialisation(); // Skip calculating the derivative
for this frame
                searchTargetSet = true; // Skips a potential problem on
the first frame
            }
            ApplyMovement(searchingTarget); // Calculate the new movement
to the target
            Searching(); // Search at the new position
        }
    }
    lastPosition = rb.transform.position; // Store a copy of the current
position
}

protected override void ApplyMovement(Vector3 target)
{
}

```

```

        float movementX = PIDUpdate(Time.fixedDeltaTime,
rb.transform.position.x, target.x, ref lastPosition.x, ref lastError.x, ref
storedIntegral.x);
        float movementY = PIDUpdate(Time.fixedDeltaTime,
rb.transform.position.y, target.y, ref lastPosition.y, ref lastError.y, ref
storedIntegral.y);
        float movementZ = PIDUpdate(Time.fixedDeltaTime,
rb.transform.position.z, target.z, ref lastPosition.z, ref lastError.z, ref
storedIntegral.z);
        Vector3 movementTotal = new Vector3(movementX, movementY, movementZ);
// Combine the movement
        if (movementTotal.magnitude > MaxMoveSpeed)
        { // Limit the movement
            movementTotal *= MaxMoveSpeed / movementTotal.magnitude; // set
movementTotal's magnitude to the maxMoveSpeed
        }
        Debug.DrawRay(rb.transform.position, new Vector3(movementX, 0f,
0f).normalized, Color.red); // X direction
        Debug.DrawRay(rb.transform.position, new Vector3(0f, movementY,
0f).normalized, Color.green); // Y direction
        Debug.DrawRay(rb.transform.position, new Vector3(0f, 0f,
movementZ).normalized, Color.blue); // Z direction
        Debug.DrawRay(rb.transform.position, movementTotal.normalized,
Color.black); // Total direction
        rb.AddForce(movementTotal, ForceMode.VelocityChange); // Apply
movement
    }

    protected override void RotateToTarget(Transform target)
    {
        Vector3 targetDirection = target.transform.position -
weapon.transform.position; // Get the direction from the weapon to the target
        Quaternion dirInQuaternion =
Quaternion.LookRotation(targetDirection.normalized, Vector3.up); // Rotation
for the next step
        rb.transform.rotation = Quaternion.Slerp(rb.transform.rotation,
dirInQuaternion, RotationSpeed * Time.fixedDeltaTime); // Rotate the enemy so
that the gun is in position to fire the player
    }
}

```

## Tank.cs

```

public class Tank : EnemyEntity
{
    public override float MaxMoveSpeed => 1f;
    public override int merit => 100;
}

```

## Soldier.cs

```
public class NewMonoBehaviourScript : EnemyEntity
{
    public override float MaxMoveSpeed => 5f;
    public override int merit => 10;
}
```

## EnemyManager.cs

```
using UnityEngine;

public class EnemyManager : MonoBehaviour
{
    public (GameObject, MemberState)[] squad; // Array of every sqad member and their state
    public enum MemberState
    {
        None,
        Searching,
        Combat,
        Dead
    }

    public void MemberDeath(GameObject member)
    {
        for (int i = 0; i < squad.Length; i++)
        { // Go through every member in the squad
            if (squad[i].Item1 == member)
            { // If the currently selected member is the one that died
                squad[i].Item2 = MemberState.Dead; // Mark it interally as dead
                break;
            }
        }
    }

    public void MemberStateChange(MemberState state)
    { // Whenever a member changes state, alert the other members
    }
}
```

## AudioManager.cs

```
using UnityEngine;
```

```

using UnityEngine.Audio;

public class AudioManager : MonoBehaviour
{
    public static AudioManager Instance;

    [SerializeField] private AudioSource sfxPrefab;
    [SerializeField] private AudioMixer mixer;

    void Awake()
    {
        if (Instance == null)
        { // Set the static reference to this
            Instance = this;
        }
    }

    public void PlaySFXClip(AudioClip clip, Transform location, float volume)
    {
        AudioSource source = Instantiate(sfxPrefab, location.position,
Quaternion.identity); // Make a new prefab at the location specified with no
rotation
        source.clip = clip; // Set the clip to start playing
        source.volume = volume; // Set the volume of the clip
        source.Play(); // Allow the clip to start playing
        Destroy(source.gameObject, source.clip.length); // Destroy the object
once the clip ends
    }
    public void SetMasterVolume(float volume)
    { // Set the volume of the Master channel
        mixer.SetFloat("masterVolume", volume);
    }
    public void SetSFXVolume(float volume)
    { // Set the volume of the SFX channel
        mixer.SetFloat("sfxVolume", volume);
    }
}

```

## DialogueDatabaseManager.cs

```

using UnityEngine;
using System.Data;
using Mono.Data.Sqlite;
using System.IO;

public class DialogueDatabaseManager : MonoBehaviour
{
    public static DialogueDatabaseManager Instance;

```

```

void Awake()
{
    if (Instance == null)
    { // If the instance has not been assigned
        Instance = this; // Set it as this one
    }
}

public IDbConnection OpenDialogueDb()
{ // Open the database with the dialogue table
    return OpenDb("Dialogue", new string[] { "english TEXT" });
}

public IDbConnection OpenActorDb()
{ // Open the database with the actor table
    return OpenDb("Actor", new string[] { "actor TEXT" });
}

private IDbConnection OpenDb(string tableName, string[] columns)
{
    string database = "URI=file:" +
Path.Combine(Application.streamingAssetsPath, "Dialogue.sqlite"); // Get a
path to the database
    IDbConnection connection = new SqliteConnection(database); // Create a
connection to it
    connection.Open(); // Open the connection
    IDbCommand createTable = connection.CreateCommand(); // Begin creating
a new command

    string columnText = ""; // Store the creation text for the columns
    foreach (string column in columns)
    { // For each column supplied
        columnText += column + ", "; // Add it to columnText
    }
    columnText = columnText[..^2]; // Remove the trailing ', '
    createTable.CommandText = $"CREATE TABLE IF NOT EXISTS {tableName} (id
INTEGER PRIMARY KEY, {columnText})"; // Set the SQL command
    createTable.ExecuteReader(); // Execute the query

    return connection;
}
}

```

## DialogueManager.cs

```

using System.Collections;
using System.Collections.Generic;
using TMPro;
using UnityEngine;

```

```
public class DialogueManager : MonoBehaviour
{
    public static DialogueManager Instance;
    public Canvas subtitleCanvas;
    public TMP_Text textObj;
    public float fadeAwayTimer = 2;
    public int maxLines = 5;
    public List<string> lines = new List<string>();
    // Start is called once before the first execution of Update after the
    MonoBehaviour is created
    void Start()
    {
        if (Instance == null)
        { // Assign the static reference to this instance
            Instance = this;
        }
    }
    void Update()
    {
        if (lines.Count == 0)
        { // If no lines are shown
            subtitleCanvas.GetComponent<CanvasGroup>().alpha = 0; // Hide the
            canvas
            return;
        }
        string text = "";
        foreach (string line in lines)
        { // For every line
            text += line + "\n"; // Construct the line that goes in the text
            object
        }
        textObj.text = text; // Assign the text inside
    }
    public void DisplayLine(string text)
    {
        Debug.Log($"Sent text: '{text}'");
        subtitleCanvas.GetComponent<CanvasGroup>().alpha = 1; // Show the
        subtitle box
        if (lines.Count >= maxLines)
        { // If too many lines are shown
            lines.Remove(lines[0]); // Remove the starting one
            StopCoroutine(Timer(lines[0])); // Stop the timer for that piece
            of text
        }
        lines.Add(text); // Add the new line
        StartCoroutine(Timer(text)); // Start a fade out time for that line
    }
    private IEnumerator Timer(string text)
```

```
{  
    yield return new WaitForSeconds(fadeAwayTimer); // Wait some seconds  
    before the line is removed  
    lines.Remove(text); // Remove the line  
}  
}
```

## MenuManager.cs

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using UnityEngine;  
using UnityEngine.SceneManagement;  
  
public class MenuManager : MonoBehaviour  
{  
    public static MenuManager Instance;  
    public PlayerManager manager;  
  
    public Canvas main;  
    public Canvas slots;  
    public Canvas options;  
    public Canvas pause;  
    public Canvas hud;  
    public Canvas death;  
  
    public string currentMenu = "";  
    public string lastMenu = "";  
  
    #nullable enable  
    [SerializeField]  
    public Dictionary<string, (Canvas, bool)> currentCanvas;  
  
    #nullable disable  
  
    void Awake()  
    {  
        if (Instance == null)  
        {  
            Instance = this; // Sets the menu manager to this object  
        }  
        currentCanvas = new Dictionary<string, (Canvas, bool)>()  
        { // Assign the canvas and the interactability of each canvas  
            { "Main", (main, true) },  
            { "Slots", (slots, true) },  
            { "Pause", (pause, true) },  
            { "HUD", (hud, false) },  
        };  
    }  
}
```

```

        { "Options", (options, true) },
        { "Death", (death, true) },
    };
}

void Start()
{
    manager = PlayerManager.Instance; // Grab the player manager
    FetchCanvases(new Scene(), new LoadSceneMode()); // Fetch all the
canvases
    SceneManager.sceneLoaded += FetchCanvases; // Attach the fetching of
the canvases to every scene load
    DisableAll(); // Disable every canvas
    ChangeMenu(manager.GetCanvasOpen(SceneManager.GetActiveScene().name));
// Change the canvas
}
public void FetchCanvases(Scene _, LoadSceneMode __)
{
// Attempt to fetch a reference to every canvas
    foreach (string canvas in currentCanvas.Keys.ToList())
    {
// get a copy of the keys
        if (currentCanvas[canvas].Item1 == null)
        {
// If there is no canvas object
            try
            {
                currentCanvas[canvas] =
                    GameObject.Find(canvas).GetComponent<Canvas>(),
                    currentCanvas[canvas].Item2
            }; // try to find the canvas and assign it if so
        }
        catch (NullReferenceException e)
        {
// If it doesn't exist, log the error
            Debug.LogError("Caught exception: " + e.Message);
            continue;
        }
    }
}
Debug.Log("Fetched all canvases");
//ChangeMenu(manager.GetCanvasOpen(SceneManager.GetActiveScene().name))
};

public void DisableAll()
{
    foreach ((Canvas, bool) canvas in currentCanvas.Values.ToList())
    {
// get a copy of the values
        if (canvas.Item1 != null)
        {
// If the canvas exists
            canvas.Item1.GetComponent<CanvasGroup>().interactable = false;
        }
    }
}

```

```
// Make the player unable to interact with the menu
    canvas.Item1.GetComponent<CanvasGroup>().blocksRaycasts =
false; // Make it invisible to all inputs
    canvas.Item1.GetComponent<CanvasGroup>().alpha = 0; // Make it
invisible to the player
}
}

Debug.Log("Disabled all canvases");
}

public void ChangeMenu(string menu)
{
    if (!currentCanvas.Keys.Contains(menu)){ // If the menu is not in the
dictionary keys
        Debug.LogError($"The menu '{menu}' is not in the registered list
of accepted menus");
    }
    if (currentMenu != "" && currentCanvas[currentMenu].Item1 != null)
    { // Disable the current menu if it exists
        currentCanvas[currentMenu].Item1.GetComponent<CanvasGroup>().inter
actable = false; // Make the player unable to interact with the menu
        currentCanvas[currentMenu].Item1.GetComponent<CanvasGroup>().block
sRaycasts = false; // Make it invisible to all inputs
        currentCanvas[currentMenu].Item1.GetComponent<CanvasGroup>().alpha
= 0; // Make it invisible to the player
        lastMenu = currentMenu; // Sets the previous menu to the current
one
    } // Enable the new canvas
    currentCanvas[menu].Item1.GetComponent<CanvasGroup>().interactable =
currentCanvas[menu].Item2; // Make it interactable if it is supposed to be
    currentCanvas[menu].Item1.GetComponent<CanvasGroup>().blocksRaycasts =
currentCanvas[menu].Item2; // Make it possible for inputs to sense it if you
should be able to interact with it
    currentCanvas[menu].Item1.GetComponent<CanvasGroup>().alpha = 1; // /
Make it visible to the player
    currentMenu = menu; // Changes the current menu
    Debug.Log($"Changed menu to '{menu}'");
    manager.SetCursorMode();
}

public void GoBack()
{
    if (lastMenu != "" && currentCanvas[lastMenu].Item1 != null)
    { // If there is a previous menu and it exists in the scene.
        ChangeMenu(lastMenu); // change to the previous menu
    }
    else
    {
}
```

```

        Debug.LogWarning("No back menu option set!");
    }

    public bool GetCurrentCanvasInteractivity()
    {
        return currentCanvas[currentMenu].Item2;
    }
}

```

## PlayerManager.cs

```

using Mono.Data.Sqlite;
using System;
using System.Data;
using System.IO;
using System.Reflection;
using UnityEngine;
using UnityEngine.AddressableAssets;
using UnityEngine.ResourceManagement.AsyncOperations;
using UnityEngine.SceneManagement;

public class PlayerManager : MonoBehaviour
{
    public static PlayerManager Instance;

    public GameObject player;
    public LevelLoading sceneLoader;
    public MenuManager menuManager;

    private IDbConnection db;

    public PlayerInputs inputs;
    public SaveData saveData;
    public int merit = 0;
    public float sanity = 100f;
    public int totalKills;
    public int currentSlot = 1;

    void Awake()
    {
        if (Instance == null)
        {
            Instance = this; // Sets the player manager to this object
        }
        inputs = new PlayerInputs(); // make a new set of player inputs
        inputs.Menu.Enable(); // Enabling only the Menu input map
    }
}

```

```
        saveData = new SaveData(); // make a new empty saveData
    }

    void Start()
    {
        menuManager = MenuManager.Instance; // Get the menu manager
        DontDestroyOnLoad(gameObject); // Since the game will start in the
title screen, it makes sure that the player, manager, and all things attached
are loaded and maintained across all scene loads
        //saveData.playerData.player = player;
        //SceneManager.sceneLoaded += ApplyData;
        SceneManager.sceneLoaded += SceneLoad; // Attach the relevant
functions to the new scene load
        //FetchReferences();
        //SceneManager.sceneLoaded += FetchPlayer;
    }

    public void SaveGame(string type)
    {
        saveData.UpdateSaveData(player); // Update the save data
        SaveSystem.Save(saveData, type, currentSlot); // Write a save file
with the current save data
    }

    void ApplyData()
    { // When a scene loads, it reloads the save data
        player.GetComponent<PlayerEntity>().Health =
saveData.playerData.health; // Apply the player's health
        player.GetComponent<PlayerEntity>().MaxHealth =
saveData.playerData.maxHealth; // Apply the player's maximum health
        player.transform.position = saveData.playerData.position; // Apply the
player's position
        player.GetComponent<Rigidbody>().linearVelocity =
saveData.playerData.velocity; // Apply the player's velocity
        player.transform.rotation = saveData.playerData.bodyRotation; // Apply
the player's body's rotation
        Camera.main.transform.rotation = saveData.playerData.lookRotation; // Apply
the player's vertical looking direction
        // Grab the prefab in the right order for the weapons
        foreach (WeaponData data in saveData.playerData.weaponSlots)
        {
            Addressables.LoadAssetAsync<GameObject>(data.name).Completed +=
(asyncOp) =>
{
    { // Load the weapon based on the ID
        if (asyncOp.Status == AsyncOperationStatus.Succeeded)
        { // If the load succeeded
            GameObject weapon = Instantiate(asyncOp.Result); // Create
the weapon
        }
    }
}
    }
}
```

```

        player.GetComponent<PlayerGunInteraction>().AddWeapon(weapon);
    // Add the weapon to the player
        weapon.GetComponent<Weapon>().TotalAmmo = data.totalAmmo;
    // Set the weapon's total ammo
        weapon.GetComponent<Weapon>().AmmoInClip =
data.ammoInClip; // Set the remaining ammo in the weapon's clip
        weapon.GetComponent<Weapon>().ClipCapacity =
data.clipCapacity; // Set the maximum capacity of the clip in the weapon
    }
    else
    {
        Debug.LogError("Failed to load weapon");
    }
};

}

player.GetComponent<PlayerGunInteraction>().activeWeaponSlot =
saveData.playerData.activeWeaponSlot; // Set the active weapon
merit = saveData.playerData.merit; // Set the merit the player has
sanity = saveData.playerData.sanity; // Set the player's sanity level
totalKills = saveData.playerData.totalKills; // Set the player's total
kills
}

public void SetSlotandData(int slot, SaveData data)
{
    currentSlot = slot; // Assign the slot to load
    saveData = data; // Assign the save data
}

public void LoadSlot(string sceneName)
{
    if(Application.CanStreamedLevelBeLoaded(sceneName)){ // Check if the
level exists
        menuManager.DisableAll();
        sceneLoader.gameObject.SetActive(true); // Enables the load screen
        sceneLoader.Load(sceneName); // Once the new scene has loaded,
execution will return here
    }
    else
    {
        Debug.LogError($"Scene '{sceneName}' does not exist.");
    }
}

public void LoadGame()
{
    // Used after a death to call SceneLoad
    SceneLoad(SceneManager.GetActiveScene(), new LoadSceneMode()); // Wrapper
}

```

```
    }

    public void LoadNextLevel(string sceneName)
    {
        SaveGame("Auto"); // Make an automatic save in case the level crashes
        the game
        LoadSlot(sceneName); // Load the next scene
    }

(nullable enable
    public int GetVarValue(string variableName)
    { // Gets the value of the variable named within itself from the world
    flags
        Type type = typeof(WorldFlags); // Gets the id of the type of
    WorldFlags
        PropertyInfo? property = type.GetProperty(variableName); //
    Potentially null output from trying to find the variable
        if (property == null)
        {
            Debug.LogError($"{variableName} is not a variable of the
    manager.");
            return -1; // Error code
        }
        else
        {
            return Convert.ToInt32(property.GetValue(saveData.worldFlags,
    null)); // Cast to an integer
        }
    }

    public void SetVarValue(string variableName, int value)
    { // Sets the value of the variable named within itself
        Type type = typeof(WorldFlags); // Gets the id of the type of
    WorldFlags
        PropertyInfo? property = type.GetProperty(variableName); //
    Potentially null output from trying to find the variable
        if (property == null)
        {
            Debug.LogError($"{variableName} is not a variable of the
    manager.");
            // Do nothing to not cause havoc
        }
        else
        {
            property.SetValue(saveData, value); // save the changed value
        }
    }

(nullable disable
```

```
public void AddMerit(int amount) => merit += amount; // Increment the  
merit by amount. If amount is negative, allows subtracting merit as well  
  
private IDbConnection OpenDb()  
{ // Opens the database and returns a connection to it  
    string database = "URI=file:" +  
Path.Combine(Application.streamingAssetsPath, "SceneInfo.sqlite"); // The  
exact file path to where it is stored  
    IDbConnection connection = new SqliteConnection(database); // Make a  
new connection with it  
    connection.Open(); // Open the connection  
    IDbCommand createTable = connection.CreateCommand(); // Begins a query  
    createTable.CommandText = $"CREATE TABLE IF NOT EXISTS SceneInfo (name  
TEXT PRIMARY KEY NOT NULL, moveability INTEGER DEFAULT 1, startX REAL DEFAULT  
0, startY REAL DEFAULT 0.85001, startZ REAL DEFAULT 0, maxEnemyCount INTEGER  
DEFAULT 0, menuOpen TEXT DEFAULT 'HUD')"; // Verifies that the required table  
exists, and creates it if not  
    createTable.ExecuteReader(); // Execute the query  
    return connection;  
}  
  
private (IDataReader, bool) QueryDatabase(string sceneName, string field)  
{  
    db = OpenDb(); // Open and store a reference to the database  
    IDbCommand request = db.CreateCommand(); // Begins a query  
    request.CommandText = $"SELECT {field} FROM SceneInfo WHERE  
name='{sceneName}'"; // Get whether the player's movement script should be  
enabled or not  
    IDataReader response = request.ExecuteReader(); // Execute the query  
    bool success = response.Read(); // Read if the query succeeded  
    if (!success)  
    {  
        Debug.LogWarning($"Scene '{sceneName}' does not have an entry in  
the info database for the player's {field}.");  
    }  
    return (response, success); // Most of the time the scene will be a  
level, so it is safer to put true.  
}  
  
public bool GetMoveability(string sceneName)  
{  
    (IDataReader response, bool success) = QueryDatabase(sceneName,  
"moveability"); // Query for the moveability  
    bool output = success ? response.GetInt32(0) != 0 : true; // return  
whether the first integer result is equal to 0 if succeeded, otherwise return  
true since it will usually be a level  
    db.Close(); // Close the database
```

```
        return output;
    }

    public string GetCanvasOpen(string sceneName)
    {
        (IDataReader response, bool success) = QueryDatabase(sceneName,
"menuOpen"); // Query for the open canvas
        string output = success ? response.GetString(0) : "HUD"; // Return
the first string result if succeeded, otherwise return "HUD" since it will
usually be a level
        db.Close(); // Close the database
        return output;
    }

    private Vector3 GetStartPos(string sceneName)
    {
        (IDataReader response, bool success) = QueryDatabase(sceneName,
"startX, startY, startZ"); // Query for the start position
        Vector3 output = success ? new Vector3(response.GetFloat(0),
response.GetFloat(1), response.GetFloat(2)) : new Vector3(0, 0.85f, 0); // //
Return the combination of the 3 axis if succeeded or the database default
otherwise
        db.Close(); // Close the database
        return output;
    }

    void SceneLoad(Scene scene, LoadSceneMode __)
    {
        sceneLoader.gameObject.SetActive(false);
        if (GetMoveability(scene.name))
        { // If the player should move
            inputs.Player.Enable(); // Enable movement
            inputs.Menu.Disable(); // Disable menu inputs
        }
        else
        {
            inputs.Player.Disable(); // Disable movement
            inputs.Menu.Enable(); // Enable menu inputs
        }
        player.transform.position = GetStartPos(scene.name); // Grab the
player's start position

        Debug.Log("Requested canvas on scene load");
        menuManager.ChangeMenu(GetCanvasOpen(scene.name)); // Change the menu
to the one that should be open
        player.GetComponent<PlayerMovement>().movementState =
PlayerMovement.PlayerMovementState.idle; // Give the player the ability to
move
    }
}
```

```

        ApplyData(); // Reload the save data
    }

    public void PlayerDeath()
    {
        player.GetComponent<PlayerMovement>().movementState =
PlayerMovement.PlayerMovementState.dead; // Stop the player's movement
    }

    public void SetCursorMode()
    {
        Camera.main.GetComponent<PlayerLook>().SetCursorLock(SceneManager.GetA
ctiveScene(), new LoadSceneMode()); // Set whether the cursor should be locked
or not
    }

    public void Quit()
    {
#if UNITY_EDITOR // If this function is being run inside the Unity Editor
        UnityEditor.EditorApplication.isPlaying = false; // Set the current
playing state to false
#else
        Application.Quit(); // Call the system-level quit on the game
#endif
    }
}

```

## Options.cs

```

using UnityEngine;
using UnityEngine.UI;

public class Options : MonoBehaviour
{
    [SerializeField] private Slider volume;
    [SerializeField] private Slider fov;
    [SerializeField] private Slider zoomFov;

    public void UpdateVolume()
    {
        AudioManager.Instance.SetMasterVolume(Mathf.Lerp(-80, 0,
volume.value)); // Set the master volume to the value of the slider
    }
    public void UpdateNormalFOV()
    {
        PlayerManager.Instance.player.GetComponent<PlayerGunInteraction>().fov
= fov.value; // Set the normal FOV
    }
}

```

```

public void UpdateZoomedFOV()
{
    PlayerManager.Instance.player.GetComponent<PlayerGunInteraction>().zoomFov = zoomFov.value; // Set the zoomed in FOV
}
}

```

## PauseGame.cs

```

using UnityEngine;
using UnityEngine.InputSystem;
using UnityEngine.SceneManagement;

public class PauseGame : MonoBehaviour
{
    private PlayerManager manager;
    private PlayerInputs playerInputs;
    bool paused = false;

    void Start()
    {
        manager = PlayerManager.Instance; // Get the game manager
        playerInputs = manager.inputs; // Get the player's inputs
        playerInputs.Player.Pause.performed += Pause; // Run Pause() when the
player pauses the game
        playerInputs.Menu.Exit.performed += Pause; // Run Pause() when the
player unpauses the game
    }

    private void Pause(InputAction.CallbackContext context)
    {
        if (paused)
        { // If the game is paused
            playerInputs.Player.Enable(); // Enable the normal inputs
            playerInputs.Menu.Disable(); // Disable the menu inputs
            manager.menuManager.ChangeMenu("HUD"); // Change to the HUD
            Time.timeScale = 1; // Resume the game
            paused = false; // The game is no longer paused
        }
        else
        {
            playerInputs.Menu.Enable(); // Enable the menu inputs
            playerInputs.Player.Disable(); // Disable the normal inputs
            manager.menuManager.ChangeMenu("Pause"); // Change to the pause
menu
            Time.timeScale = 0; // Pause the game
            paused = true; // The game is currently paused
        }
    }
}

```

```
    }  
}
```

## BodyDetection.cs

```
using UnityEngine;  
  
public class BodyDetection : MonoBehaviour  
{  
    public PlayerMovement bodyScript;  
  
    private void OnCollisionStay(Collision collision)  
    { // When the body touches something  
        bodyScript.CollisionDetected(collision); // Send all collision data to  
the movement script  
    }  
    private void OnCollisionExit(Collision collision)  
    { // When the body stops touching something  
        bodyScript.CollisionDetected(collision); // Send all collision data to  
the movement script  
    }  
}
```

## PlayerGunInteraction.cs

```
using System.Collections.Generic;  
using UnityEngine;  
using UnityEngine.InputSystem;  
  
public class PlayerGunInteraction : MonoBehaviour  
{  
    Camera playerHead;  
    public Transform weaponHold;  
    public Transform weaponStow;  
    // private Vector3 difference;  
    public PlayerManager manager;  
    public List<GameObject> weaponSlots = new List<GameObject>(); // Maybe  
reimplement as a list, so that scrolling works  
    public int activeWeaponSlot = 0;  
    public int maxWeapons = 4;  
    private PlayerInputs playerInputs;  
    private PlayerLook lookScript;  
    private LayerMask notPlayerLayer;  
  
    [Header("Zoom")]  
    public float fov = 60;  
    public float zoomFov = 20;
```

```
[SerializeField] private float rotationSpeed = 10f;

/*
This script should handle swapping between weapon slots, picking up
weapons, firing the weapons
*/
private void Start()
{
    manager = PlayerManager.Instance; // Get the game manager
    playerInputs = manager.inputs; // Get the player's inputs
    playerHead = Camera.main; // Get the camera
    lookScript = Camera.main.GetComponent<PlayerLook>(); // Get the
player's looking script
    notPlayerLayer = ~LayerMask.GetMask("Player"); // Get the layer that
the player isn't on
    playerInputs.Player.Fire.performed += Fire; // Run Fire() when the
player presses the fire key
    playerInputs.Player.Reload.performed += Reload; // Run Reload() when
the player presses the reload key
    playerInputs.Player.SwapWeapon.performed += SwapWeapon; // Run
SwapWeapon() when the player wants to swap the weapon
    playerInputs.Player.WeaponSlot1.performed += (i) =>
SwapSpecificSlot(1); // Swap to slot 1
    playerInputs.Player.WeaponSlot2.performed += (i) =>
SwapSpecificSlot(2); // Swap to slot 2
    playerInputs.Player.WeaponSlot3.performed += (i) =>
SwapSpecificSlot(3); // Swap to slot 3
    playerInputs.Player.WeaponSlot4.performed += (i) =>
SwapSpecificSlot(4); // Swap to slot 3
    playerInputs.Player.Interact.performed += Interact; // Run Interact()
when the player presses the interact key

    foreach (GameObject weapon in weaponSlots)
    { // for every equipped weapon
        if (weapon != null)
        { // If it exists
            weapon.SetActive(false); // Disable it
            weapon.GetComponent<Rigidbody>().isKinematic = true; //
Disable the physics system for it
            weapon.GetComponent<Rigidbody>().detectCollisions = false; //
Disable its ability to detect collision
            weapon.transform.position = weaponStow.position; // Move the
weapon to the stowed position
        }
    }
    if (weaponSlots.Count > 0)
    { // If the player has more than 1 weapon
```

```

        weaponSlots[0].SetActive(true); // Activate the first one
        weaponSlots[0].transform.position = weaponHold.position; // Place
it into the held position
    }
}

private void Update()
{
    Zoom(); // Check the fov state
    if (activeWeaponSlot >= maxWeapons || activeWeaponSlot < 0)
    { // if the current weapon slot is outside the maximum weapon range
        activeWeaponSlot = Wrap(activeWeaponSlot, weaponSlots); // Wrap it
appropriately
    }
    if (weaponSlots.Count > 0)
    { // If there is a weapon active
        RaycastHit playerHit;
        Quaternion lookRotation = playerHead.transform.rotation; // temporarily store the current facing rotation
        if (Physics.Raycast( // if this raycast hits something
            origin: playerHead.transform.position,
            direction: playerHead.transform.rotation * Vector3.forward,
            hitInfo: out playerHit,
            layerMask: notPlayerLayer,
            maxDistance: Mathf.Infinity
        ))
        {
            Vector3 pointDirection = (playerHit.point -
weaponSlots[activeWeaponSlot].transform.position).normalized; // find the
direction the weapon needs to point towards
            lookRotation = Quaternion.LookRotation(pointDirection); // Store the required rotation to reach the target
        }
        weaponSlots[activeWeaponSlot].transform.rotation =
Quaternion.Slerp(weaponSlots[activeWeaponSlot].transform.rotation,
lookRotation, rotationSpeed * Time.deltaTime); // smoothly turn the weapon
towards the
    }
}

private GameObject SelectFireWeapon()
{ // Get the currently selected weapon
    return weaponSlots[activeWeaponSlot];
}

private int Wrap(int value, List<GameObject> list)
{
    return ((value % list.Count) + list.Count) % list.Count; // Wrap both
around negatively and positively
}

private void Fire(InputAction.CallbackContext inputType)

```

```
{  
    if (weaponSlots.Count > 0)  
    { // Checks that there is a weapon to fire  
        GameObject weapon = SelectFireWeapon(); // Get the current weapon  
        weapon.GetComponent<Gun>().Fire(); // Fire it  
    }  
}  
private void Reload(InputAction.CallbackContext inputType)  
{  
    if (weaponSlots.Count > 0)  
    { // Checks that there is a weapon to reload  
        GameObject weapon = SelectFireWeapon(); // Get the current weapon  
        weapon.GetComponent<Gun>().Reload(); // Reload it  
    }  
}  
private void Zoom()  
{  
    if (playerInputs.Player.Zoom.inProgress)  
    { // If currently pressing zoom  
        Camera.main.fieldOfView = zoomFov; // Set the camera fov to the  
        zoomed in one  
        lookScript.sensitivity = lookScript.zoomSensitivity; // Decrease  
        the camera sensitivity  
    }  
    else  
    {  
        Camera.main.fieldOfView = fov; // Set the camera fov to the normal  
        one  
        lookScript.sensitivity = lookScript.normalSensitivity; // Return  
        the camera sensitivity to normal  
    }  
}  
private void SwapWeapon(InputAction.CallbackContext inputType)  
{  
    if (weaponSlots.Count > 0)  
    { // If the player has weapons  
        int direction = (int)inputType.ReadValue<float>(); // Get the swap  
        direction and cast it to an integer  
        weaponSlots[activeWeaponSlot].SetActive(false); // Disable the  
        current weapon  
        weaponSlots[activeWeaponSlot].transform.position =  
        weaponStow.position; // Move to the stowed position  
        activeWeaponSlot = Wrap(activeWeaponSlot + direction,  
        weaponSlots); // Get the new slot  
        weaponSlots[activeWeaponSlot].transform.position =  
        weaponHold.position; // Move that weapon to the held position  
        weaponSlots[activeWeaponSlot].SetActive(true); // Enable this  
        weapon
```

```

        }

    private void SwapSpecificSlot(int slot)
    {
        if (slot > weaponSlots.Count)
        { // Don't swap if there aren't that many weapons to swap to
            return;
        }
        if (weaponSlots[slot] == null)
        { // If that weapon doesn't exist don't swap to it
            return;
        }
        weaponSlots[activeWeaponSlot].SetActive(false); // Disable the current
        weapon
        weaponSlots[activeWeaponSlot].transform.position =
        weaponStow.position; // Move to the stowed position
        activeWeaponSlot = slot; // Swap to the specific slot
        weaponSlots[activeWeaponSlot].transform.position =
        weaponHold.position; // Move that weapon to the held position
        weaponSlots[activeWeaponSlot].SetActive(true); // Enable this weapon
    }

    private int GetSwapSlot(out bool wasUsed)
    {
        if (weaponSlots.Count == 0){ // If there are no weapons
            wasUsed = false;
            return 0;
        }
        if (weaponSlots.Count < maxWeapons)
        { // If there are places left
            wasUsed = false;
            return activeWeaponSlot + 1;
        }
        wasUsed = true;
        return activeWeaponSlot; // Return the current slot if there isn't an
        empty one
    }

    private void Interact(InputAction.CallbackContext inputType)
    {
        RaycastHit hit;
        Debug.DrawLine(playerHead.transform.position,
        playerHead.transform.position + playerHead.transform.rotation *
        Vector3.forward, Color.red, 1f);
        if (Physics.Raycast(origin: playerHead.transform.position, direction:
        playerHead.transform.rotation * Vector3.forward, hitInfo: out hit))
        { // If the raycast collides with something
            Debug.Log("Raycast hit " + hit.collider.gameObject.name);
            if (hit.collider.gameObject.tag == "Interactable")
            { // If the hit object is interactable

```

```
        InteractHit(hit.collider.gameObject); // Interact with it
    }
    else if (hit.collider.gameObject.tag == "Weapon")
    { // If the hit object is a weapon
        AddWeapon(hit.collider.gameObject); // Add the weapon to the
player's slots
    }
}
public void AddWeapon(GameObject hit)
{
    Debug.Log("weapon interaction");
    bool wasUsed;
    int temp = GetSwapSlot(out wasUsed); // Check which slot to swap into
    if (wasUsed)
    { // remove the current weapon
        weaponSlots[activeWeaponSlot].transform.SetParent(null, true); // Free the weapon from the player
        weaponSlots[activeWeaponSlot].GetComponent<Rigidbody>().isKinematic = false; // Allow the physics system to push it
        weaponSlots[activeWeaponSlot].GetComponent<Rigidbody>().detectCollisions = true; // Allow it to detect collisions
    }
    else if (!weaponSlots.Contains(hit))
    { // add the new weapon
        weaponSlots.Add(hit);
    }
    activeWeaponSlot = temp; // Set the new weapon slot
    // Format the new weapon to the appropriate locations and properties
    weaponSlots[activeWeaponSlot].transform.SetParent(playerHead.transform, true); // Parent the weapon to the camera
    weaponSlots[activeWeaponSlot].GetComponent<Rigidbody>().isKinematic = true; // Disable the physics system from interacting with it
    weaponSlots[activeWeaponSlot].GetComponent<Rigidbody>().detectCollisions = false; // Don't allow it to detect collisions
    weaponSlots[activeWeaponSlot].transform.position =
    weaponHold.position; // Put the weapon in the held position
    weaponSlots[activeWeaponSlot].transform.rotation =
    weaponHold.rotation; // Rotate the weapon properly
}
private void InteractHit(GameObject hit)
{
    hit.GetComponent<Interactable>().Interact(); // Call Interact() on the interactable
}
```

# PlayerLook.cs

```
using UnityEditor;
using UnityEngine;
using UnityEngine.InputSystem;
using UnityEngine.SceneManagement;

public class PlayerLook : MonoBehaviour
{
    public PlayerManager manager;
    private InputAction lookAction;
    [SerializeField] private Rigidbody playerBody;
    private float verticalRotation = 0f;
    public float sensitivity = 1;
    public float normalSensitivity = 1;
    public float zoomSensitivity = 0.5f;

    void Start()
    {
        manager = PlayerManager.Instance; // Fetch a reference to the game
manager
        lookAction = manager.inputs.Player.Look; // Fetch a reference to the
looking input for quick access
        //gameObject.transform.SetParent(playerBody.transform, true);
        SceneManager.sceneLoaded += SetCursorLock; // Run SetCursorLock()
every time a scene loads
        //Cursor.lockState = CursorLockMode.Locked;
        SetCursorLock(SceneManager.GetActiveScene(), new LoadSceneMode()); //
    }

    // Update is called once per frame
    void Update()
    {
        Look();
    }
    void Look()
    {
        Vector2 lookMotion = lookAction.ReadValue<Vector2>() * sensitivity /
20; // Work out the effective motion of the camera
        playerBody.rotation = playerBody.rotation *
Quaternion.Euler(Vector3.up * lookMotion.x); // Apply the body rotation

        verticalRotation -= lookMotion.y; // Work out the total vertical
rotation
        verticalRotation = Mathf.Clamp(verticalRotation, -90f, 90f); // Clamp
the output to stop the player looking upside down
        transform.localRotation = Quaternion.Euler(verticalRotation, 0f, 0f);
// Apply the camera rotation
    }
}
```

```

public void SetCursorLock(Scene scene, LoadSceneMode _)
{
    if (manager.GetMoveability(scene.name) &&
!MenuManager.Instance.GetCurrentCanvasInteractability())
        { // If the player can move, and the current canvas isn't interactable
            Cursor.lockState = CursorLockMode.Locked; // Lock the player's
cursor
            Debug.Log("Locked cursor.");
        }
    else
    {
        Cursor.lockState = CursorLockMode.None; // Unlock the player's
cursor
        Debug.Log("Unlocked cursor.");
    }
}
}

```

## PlayerMovement.cs

```

using UnityEngine;
using UnityEngine.InputSystem;

public class PlayerMovement : MonoBehaviour
{
    public PlayerManager manager;
    private PlayerInputs playerInputs; // Use if using the C# Class
    // private PlayerInput playerInput; // Use if using the Unity Interface

    [Header("Character Dimensions")]
    [SerializeField] public float playerRadius = 0.5f;
    [SerializeField] public float standingHeight = 1.7f;
    [SerializeField] public float crouchingHeight = 1f;

    [Header("Ground Checks")]
    [SerializeField] private bool isGrounded = false;
    [SerializeField] private LayerMask playerLayer;
    [SerializeField] public float groundDistance = 0.03f;
    [SerializeField] public float maxSlope;
    [SerializeField] public Vector3 groundNormal = Vector3.up;

    [Header("Jumping")]
    [SerializeField] private float gravity = -9.81f;
    [SerializeField] private float jumpForce = 3.5f;
    [SerializeField] private int airJumpsTotal = 1;
    [SerializeField] private int airJumpsLeft = 1;

    [Header("Movement")]

```

```
[SerializeField] public Vector3 movement;
[SerializeField] public PlayerMovementState movementState;
[SerializeField] public PlayerMovementState lastMovementState =
PlayerMovementState.idle;
[SerializeField] private float friction = 0.9f;
[SerializeField] private float drag = 0.01f;
[SerializeField] private Vector3 velocity;
[SerializeField] private Vector3 surfaceVelocity;
[SerializeField] private Vector2 inputDirection;
[SerializeField] private float preBoostVelocity;
[SerializeField] private float baseMovementAcceleration = 10f;
[SerializeField] private float boostMultiplier = 2f;

[Header("Crouching")]
[SerializeField] private bool lastHoldCrouchState = false;
[SerializeField] private bool holdCrouch = false;
[SerializeField] private bool toggleCrouch = false;
[SerializeField] private bool wasSliding = false;
[SerializeField] private float startedSliding;
[SerializeField] private bool slideCapSet;

[Header("Wallrunning")]
[SerializeField] private Vector3 wallNormal;
[SerializeField] private bool stuckToWall;
[SerializeField] private float wallSeparationAngle = 30f;
[SerializeField] private float wallFallSpeed = 2f;
[SerializeField] private float separationBoost = 2f;
[SerializeField] private float gravityMultiplier = 0.1f;

[Header("Speed Caps")]
[SerializeField] private float maxCrouchSpeed = 4.5f;
[SerializeField] private float maxSlidingSpeed = 15f;
[SerializeField] private float maxWalkSpeed = 7.5f;
[SerializeField] private float maxSprintSpeed = 15f;
[SerializeField] private float maxBoostSpeed;
[SerializeField] private float maxWallrunningSpeed = 30f;

[Header("Objects")]
[SerializeField] private GameObject body;
[SerializeField] private Rigidbody player;
[SerializeField] private CapsuleCollider playerCollider;
const float minSpeed = 1e-4f;

public enum PlayerMovementState
{
    idle,
    crouching,
    sliding,
```

```
walking,
sprinting,
boosting,
wallrunning,
dead
}
private void Start()
{
    manager = PlayerManager.Instance;
    playerLayer = LayerMask.GetMask("Standable");

    player = body.GetComponent<Rigidbody>();
    playerCollider = body.GetComponent<CapsuleCollider>();
    SetPlayerDimensions(standingHeight);

    playerInputs = manager.inputs;

    playerInputs.Player.Jump.performed += Jump;
    playerInputs.Player.Boost.performed += Boost;

    maxSlope = GlobalConstants.maxSlope;

    movementState = PlayerMovementState.idle;
    airJumpsLeft = airJumpsTotal;
}
private void Update()
{
    if (movementState != PlayerMovementState.dead)
    { // If the player isn't dead
        velocity = player.linearVelocity; // Store the current velocity
        for ease of use
            if (movementState == PlayerMovementState.wallrunning)
                { // If the player is wallrunning
                    surfaceVelocity = Vector3.ProjectOnPlane(velocity,
                    wallNormal); // project the velocity on the wall
                }
            else
                {
                    surfaceVelocity = Vector3.ProjectOnPlane(velocity,
                    groundNormal); // project the velocity on the floor
                }
        inputDirection =
        playerInputs.Player.Movement.ReadValue<Vector2>().normalized; // Get the input
        for ease of use
            bool crouching = CrouchControlState(); // Get the current
        crouching state

        SetMovementState(crouching); // Set the current movement state
    }
}
```

```
        if (movementState == PlayerMovementState.wallrunning)
        { // If wallrunning, disable all crouching states
            holdCrouch = false;
            toggleCrouch = false;
            lastHoldCrouchState = false;
            Wallrun(); // Wallrunning specific function
        }
        else
        {
            Crouch(); // Decide if the player should change size
            Movement(); // Generic movement function
        }
        if (inputDirection.magnitude == 0 && surfaceVelocity.magnitude <
0.1f)
        { // If moving very slowly horizontally and with no input
            player.linearVelocity = new Vector3(0, velocity.y, 0); // Reset the horizontal movement
        }
        lastMovementState = movementState; // Store the current movement state
    }
    else
    {
        player.linearVelocity = Vector3.zero; // freeze the player
    }
}
private void FixedUpdate()
{
    Gravity(); // Apply gravity
    player.AddForce(movement, ForceMode.VelocityChange); // Apply total movement
    movement = Vector3.zero; // Reset movement
}
private void SetMovementState(bool isCrouched)
{
    if (stuckToWall)
    { // If on the wall
        movementState = PlayerMovementState.wallrunning;
    }
    else if (surfaceVelocity.magnitude <= maxWalkSpeed && isCrouched &&
movementState != PlayerMovementState.sliding)
    { // If crouching
        movementState = PlayerMovementState.crouching;
        wasSliding = false; // Not sliding
    }
    else if (surfaceVelocity.magnitude > maxWalkSpeed && isCrouched)
    { // If sliding
```

```

        movementState = PlayerMovementState.sliding;
        if (!wasSliding)
        { // The player has just changed state
            wasSliding = true; // Have started sliding
            startedSliding = Time.time; // Set the current time
        }
        if (slideCapSet)
        { // If the maximum sliding speed has been set
            maxSlidingSpeed = surfaceVelocity.magnitude; // Set the
current velocity as the limit
        }
    }
    else if ((surfaceVelocity.magnitude < preBoostVelocity - 1 &&
surfaceVelocity.magnitude > minSpeed && surfaceVelocity.magnitude <
maxWalkSpeed && !playerInputs.Player.Sprint.inProgress && movementState ==
PlayerMovementState.boosting) || (surfaceVelocity.magnitude > minSpeed &&
surfaceVelocity.magnitude <= maxWalkSpeed &&
!playerInputs.Player.Sprint.inProgress && movementState !=
PlayerMovementState.boosting))
        { // If the player is walking, based on if they aren't supposed to be
boosting or sprinting
            movementState = PlayerMovementState.walking;
            wasSliding = false; // Not sliding
        }
    else if ((surfaceVelocity.magnitude < preBoostVelocity - 1 &&
surfaceVelocity.magnitude > minSpeed && surfaceVelocity.magnitude <
maxSprintSpeed && playerInputs.Player.Sprint.inProgress && movementState ==
PlayerMovementState.boosting) || (surfaceVelocity.magnitude > minSpeed &&
surfaceVelocity.magnitude <= maxSprintSpeed &&
playerInputs.Player.Sprint.inProgress && movementState !=
PlayerMovementState.boosting))
        { // If the player is walking based on if they aren't supposed to be
boosting and are currently holding down the sprint key
            movementState = PlayerMovementState.sprinting;
            wasSliding = false; // Not sliding
        }
    else if (inputDirection.magnitude == 0)
    { // If no input is held and not in any of the other states
        movementState = PlayerMovementState.idle;
        wasSliding = false; // Not sliding
    }
}
private void SetPlayerDimensions(float height)
{
    float previousHeight = playerCollider.height; // Store the current
height
    playerCollider.height = height; // Set the new height
    player.transform.position -= new Vector3(0, (previousHeight - height)
}

```

```
// 2, 0); // Work out the movement distance
    }
    private bool CrouchControlState()
    {
        holdCrouch = playerInputs.Player.HoldCrouch.inProgress; // Get the
status of the hold crouch
        if (playerInputs.Player.ToggleCrouch.triggered)
        { // If the player just pressed the toggle crouch
            toggleCrouch = !toggleCrouch; // Invert the toggle crouch
        }
        if (holdCrouch)
        { // If the player is holding crouch
            lastHoldCrouchState = true;
            return true;
        }
        else
        {
            if (lastHoldCrouchState)
            { // If the player just let go of crouching
                toggleCrouch = false;
            }
            lastHoldCrouchState = false; // Store for the next iteration
            return toggleCrouch;
        }
    }
    private void Crouch()
    {
        if (lastMovementState != movementState)
        { // If the player wasn't previously crouching
            if (movementState == PlayerMovementState.crouching ||
movementState == PlayerMovementState.sliding)
                { // If the player is supposed to be short
                    SetPlayerDimensions(crouchingHeight);
                }
            else
            {
                SetPlayerDimensions(standingHeight);
            }
        }
    }
    private void Gravity()
    {
        if (stuckToWall)
        { // If wallrunning
            movement += Physics.gravity / -9.81f * gravity * gravityMultiplier
* Time.fixedDeltaTime;
        }
        else if (!isGrounded)
```

```
{ // If in the air
    movement += Physics.gravity / -9.81f * gravity *
Time.fixedDeltaTime;
}
private float FrictionMultiplier()
{
    if (!isGrounded)
    { // If in the air
        return 1 - drag;
    }
    else if (movementState == PlayerMovementState.boosting)
    { // If boosting
        return 1 - friction / 10;
    }
    else if (movementState == PlayerMovementState.sliding)
    { // If sliding
        return 1 - friction / 2;
    }
    else if (inputDirection.magnitude == 0)
    { // If the player has let go of the inputs
        return 1 - friction;
    }
    else
    { // Default friction
        return SpeedFunction(Mathf.Clamp(surfaceVelocity.magnitude, 0,
1f), 1, 1) - 1;
    }
}
private float MaxSpeed()
{
    switch (movementState)
    { // Return the correct max speed based on the current movement state
        case PlayerMovementState.crouching:
            return maxCrouchSpeed;
        case PlayerMovementState.wallrunning:
            return maxWallrunningSpeed;
        case PlayerMovementState.sliding:
            return maxSlidingSpeed;
        case PlayerMovementState.walking:
            return maxWalkSpeed;
        case PlayerMovementState.sprinting:
            return maxSprintSpeed;
        case PlayerMovementState.boosting:
            return maxBoostSpeed;
        case PlayerMovementState.idle:
            return maxWalkSpeed;
        default:
```

```
        return surfaceVelocity.magnitude;
    }
}

private float Pow4(float num)
{
    return num * num * num * num; // num ^ 4
}

private float SpeedFunction(float speed, float a, float b)
{
    return -(Pow4(speed / a) / (b * b * b)) + b; // The function for
calculating the acceleration based on the speed
}

private float CalculateAccelerationMultiplier(Vector2? speed = null)
{
    if (speed == null) // no speed is supplied
    {
        speed = surfaceVelocity; // Use the surface velocity
    }
    float clampedMagnitude = Vector2.ClampMagnitude((Vector2)speed,
MaxSpeed()).magnitude; // Stop the speed from overflowing
    float acceleration;
    switch (movementState)
    {
        case PlayerMovementState.walking: // When the player is walking
            acceleration = SpeedFunction(clampedMagnitude, 0.75f, 10);
            break;
        case PlayerMovementState.sprinting: // When the player is
sprinting
            acceleration = SpeedFunction(clampedMagnitude, 1, 15f);
            break;
        case PlayerMovementState.boosting: // When the player is boosting
            acceleration = 10f;
            break;
        case PlayerMovementState.crouching: // When the player is
crouching
            acceleration = SpeedFunction(clampedMagnitude, 0.45f, 10f);
            break;
        case PlayerMovementState.sliding: // When the player is sliding
            float currentTime = Time.time; // Get the current time
            float difference = currentTime - startedSliding; // Work out
how long the player has been sliding
            if (difference < 0.1)
            { // If it's still initial
                acceleration = -5 * difference + 20; // Apply a high
acceleration
                slideCapSet = false; // The maximum sliding limit hasn't
been set
            }
    }
}
```

```
        else
        {
            acceleration = 0.5f; // Apply a low acceleration
            slideCapSet = true; // The maximum sliding speed has been
set
        }
        break;
    case PlayerMovementState.wallrunning: // When the player is
wallrunning
        acceleration = SpeedFunction(clampedMagnitude, 1.5f, 20f);
        break;
    default:
        acceleration = 0f;
        break;
}
return acceleration + baseMovementAcceleration;
}
private void Movement()
{
    float maxSpeed = MaxSpeed(); // Get the maximum speed for that state
    float acceleration = CalculateAccelerationMultiplier(); // Get the
acceleration for that state
    Vector3 target = player.rotation * new Vector3(inputDirection.x, 0,
inputDirection.y); // Get the target
    float alignment = Vector3.Dot(surfaceVelocity.normalized,
target.normalized);

    if (surfaceVelocity.magnitude > maxSpeed)
    { // If moving faster than the maximum speed
        acceleration *= surfaceVelocity.magnitude / maxSpeed; // Reduce
the acceleration to counteract the speed overflow
    }
    target = target * maxSpeed - surfaceVelocity; // Work out how much
extra acceleration is needed

    if (target.magnitude < 0.5f) // If moving very slowly
    {
        acceleration *= target.magnitude / 0.5f;
    }
    if (alignment <= 0)
    { // If attempting to move in a wildly opposite direction
        acceleration *= 2;
    }

    target = Vector3.ProjectOnPlane(target, groundNormal).normalized; // 
Project target on the ground
    target *= acceleration; // Work out the next movement
    target -= target * FrictionMultiplier(); // take a percentage away
}
```

```

from friction
    movement += target * Time.deltaTime; // add it to the movement for the
next physics timestep
    /*
        Vector2 directionGround = new Vector2(direction.x, direction.z);
        if (directionGround.magnitude * Time.deltaTime < 0 && -
directionGround.magnitude * Time.deltaTime > surfaceVelocity.magnitude){
            Debug.Log("did this");
            direction.x = -surfaceVelocity.x/Time.deltaTime;
            direction.z = -surfaceVelocity.y/Time.deltaTime;
        }*/
    }

//todo rotate camera away from wall
private void Wallrun()
{
    Vector3 target = player.rotation * new Vector3(inputDirection.x, 0,
inputDirection.y); // Get the intended movement globally
    if (Mathf.Abs(Vector3.Angle(wallNormal, target)) <
wallSeparationAngle)
    {
        // If attempting to move away from the wall
        stuckToWall = false; // Remove from wall
        movement += wallNormal * separationBoost * Time.deltaTime; // provide a boost to be pushed off the wall
        movementState = PlayerMovementState.walking; // Set state to walking
        return;
    }
    else if (player.linearVelocity.magnitude < wallFallSpeed)
    {
        stuckToWall = false; // Remove from wall
        movement += wallNormal * separationBoost * Time.deltaTime * 0.1f;
// Provide a small boost to be kicked off the wall
        movementState = PlayerMovementState.walking; // Set state to walking
        return;
    }

    float maxSpeed = MaxSpeed(); // Get the maximum speed for that state
    float acceleration = CalculateAccelerationMultiplier(); // Get the acceleration for that state
    Vector3 wallVelocity = Vector3.ProjectOnPlane(player.linearVelocity,
wallNormal); // Could probably change at some point to surface velocity

    if (wallVelocity.magnitude > maxSpeed)
    { // If moving above the maximum wallrunning speed
        acceleration *= wallVelocity.magnitude / maxSpeed;
    }
}

```

```
        target = target * maxSpeed - wallVelocity; // Work out how much extra
acceleration is needed
        target = Vector3.ProjectOnPlane(target, wallNormal).normalized; //
Project target on the ground
        target *= acceleration; // Work out the next movement
        target -= target * FrictionMultiplier(); // take a percentage away
from friction
        movement += target * Time.deltaTime; // add it to the movement for the
next physics timestep

    }

private void Jump(InputAction.CallbackContext inputType)
{
    /*
    Vector3 direction = inputDirection == Vector2.zero ?
surfaceVelocity.normalized : player.rotation * new Vector3(inputDirection.x,
0, inputDirection.y);
    direction.y += 1;

    direction *= jumpForce;
    if (movementState == PlayerMovementState.wallrunning){
        direction += -wallNormal * jumpForce * 2;
    }
    float alignment = Vector3.Dot(surfaceVelocity.normalized,
inputDirection);
    if (alignment < 0){
        direction += -surfaceVelocity;
    }*/
    Vector3 direction = Vector3.up * jumpForce;
    if (movementState == PlayerMovementState.wallrunning)
    { // If wallrunning
        direction += wallNormal * jumpForce; // Make sure to push the
player away from the wall
        direction = direction.normalized * jumpForce; // Push with a
predetermined magnitude
        if (velocity.y < 0)
        { // If moving down
            direction.y -= velocity.y; // Move the player back upwards
        }
        movement += direction; // Add force for the next tick
    return;
    }

    if (velocity.y < 0)
    { // If the player is moving downwards, provide enough force to move
them back upwards
        direction.y -= velocity.y;
    }
}
```

```

        if (isGrounded)
        { // If on the ground
            Debug.Log("Jumped");
            movement += direction; // Jump normally
            //player.AddForce(direction, ForceMode.VelocityChange);
        }
        else if (airJumpsLeft > 0)
        { // If allowed to double jump
            movement += direction; // Apply the same movement
            //player.AddForce(direction, ForceMode.VelocityChange);
            airJumpsLeft--; // Remove 1 double jump
            Debug.Log($"Air jumped, jumps left: {airJumpsLeft}");
        }
    }

    private void Boost(InputAction.CallbackContext inputType)
    {
        movementState = PlayerMovementState.boosting; // Set state to boosting
        preBoostVelocity = surfaceVelocity.magnitude < maxSprintSpeed ?
maxSprintSpeed : surfaceVelocity.magnitude; // If the player is moving below
the maximum sprint speed, set the calculated speed to the sprinting speed,
otherwise set it to the current speed
        inputDirection =
playerInputs.Player.Movement.ReadValue<Vector2>().normalized; // Get the
player's inputs

        Vector3 boostMovement = player.rotation * new
Vector3(inputDirection.x, 0, inputDirection.y) * preBoostVelocity *
(boostMultiplier - 1); // Work out how much extra speed the player will gain
        maxBoostSpeed = velocity.magnitude + boostMovement.magnitude; // Sets
the maximum speed
        movement += boostMovement; // Adds it to the movement
    }

    public void CollisionDetected(Collision collision)
    { // This function is called externally by the body
        if (collision.contacts.Length > 0)
        { // If the player is touching something
            bool onWall = false; // Track if the player is on a wall
            isGrounded = false; // Track if the player is on a floor
            foreach (ContactPoint contact in collision.contacts)
            {
                float slopeAngle = Vector3.Angle(contact.normal, Vector3.up);
// Calculate the angle of the slope from the vertical

                airJumpsLeft = airJumpsTotal; // Resets the airjumps
                if (slopeAngle <= maxSlope)
                { // If on the ground
                    isGrounded = true;
                    groundNormal = contact.normal;
                }
            }
        }
    }
}

```

```
        break;
    }
    else
    { // On the wall
        onWall = true;
        wallNormal = contact.normal;
    }
}
if (onWall && !isGrounded)
{ // The player is wallrunning
    movementState = PlayerMovementState.wallrunning;
    stuckToWall = true;
}
else
{ // Not touching anything
    isGrounded = false; // In the air
    stuckToWall = false; // No longer on the wall
    movementState = PlayerMovementState.walking; // apply normal
walking state
    groundNormal = Vector3.up; // Reset groundNormal to default
}
}
```

# Checkpoint.cs

```
using UnityEngine;

public class Checkpoint : MonoBehaviour
{
    public PlayerManager manager;

    void Start()
    {
        manager = PlayerManager.Instance; // Get the game manager
    }

    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.tag == "Player")
        { // If the player has touched the checkpoint
            manager.SaveGame("Auto"); // Create an automatic save
            Debug.Log("Saved");
        }
    }
}
```

## LevelBullets.cs

```
[System.Serializable]
public class LevelBullets
{
    public BulletSaveData[] level1;
    public LevelBullets()
    {
        level1 = new BulletSaveData[50]; // Each level can hold a maximum of
50 bullets
    }
}
[System.Serializable]
public class BulletSaveData
{
    public SerVector3 position;
    public int timePlacedIntoSave;
    public BulletSaveData()
    { // Create empty bullet data
        position = new SerVector3(0, 0, 0);
        timePlacedIntoSave = -1;
    }
    public BulletSaveData(SerVector3 position, int currentTime)
    { // Set the data of the bullets
        this.position = position;
        timePlacedIntoSave = currentTime;
    }
}
```

## LevelHighscores.cs

```
using System.Collections.Generic;

[System.Serializable]
public class LevelHighscores
{
    public HighscoreData level1;
    public LevelHighscores()
    {
        level1 = new HighscoreData();
    }
}

[System.Serializable]
public class HighscoreData
```

```
{  
    public int time;  
    public int enemiesKilled;  
    public bool[] secretsFound;  
    public int deaths;  
    public List<string> weaponsUsed;  
    public HighscoreData()  
    { // Empty highscore data  
        time = 0;  
        enemiesKilled = 0;  
        secretsFound = new bool[5];  
        deaths = 0;  
        weaponsUsed = new List<string>();  
    }  
    public HighscoreData(int time, int enemiesKilled, bool[] secretsFound, int  
deaths, List<string> weaponsUsed)  
    { // Fill in the highscore data  
        this.time = time;  
        this.enemiesKilled = enemiesKilled;  
        this.secretsFound = secretsFound;  
        this.deaths = deaths;  
        this.weaponsUsed = weaponsUsed;  
    }  
}
```

## PlayerData.cs

```
using System.Collections.Generic;  
using Newtonsoft.Json;  
using UnityEngine;  
using UnityEngine.SceneManagement;  
  
[System.Serializable]  
public class PlayerData  
{  
    public string currentScene;  
    public float health;  
    public float maxHealth;  
    public SerVector3 position;  
    public SerVector3 velocity;  
    public SerQuaternion bodyRotation;  
    public SerQuaternion lookRotation;  
    public List<WeaponData> weaponSlots;  
    public int activeWeaponSlot;  
    public int merit;  
    public float sanity;  
    public float timeOnSave;  
    public int totalKills;
```

```
public PlayerData()
{
    // Initialise PlayerData with some default values
    currentScene = "Tutorial";
    health = 100f;
    maxHealth = health;
    position = new SerVector3(0, 0.85f, 0);
    velocity = new SerVector3(0, 0, 0);
    bodyRotation = new SerQuaternion(0, 0, 0, 0);
    lookRotation = new SerQuaternion(0, 0, 0, 0);
    weaponSlots = new List<WeaponData>();
    activeWeaponSlot = 0;
    merit = 0;
    sanity = 100f;
    timeOnSave = 0;
    totalKills = 0;
}

public void UpdateData(GameObject player)
{
    currentScene = SceneManager.GetActiveScene().name; // Get the scene
    the player is in
    health = player.GetComponent<PlayerEntity>().Health; // Get the
    player's health from their entity script
    maxHealth = player.GetComponent<PlayerEntity>().MaxHealth; // Get the
    player's maximum health from their entity script
    position = player.transform.position; // Get the player's position
    velocity = player.GetComponent<Rigidbody>().linearVelocity; // Get the
    player's current velocity from the rigidbody attached to it
    bodyRotation = player.transform.rotation; // Get the player's current
    rotation
    lookRotation = Camera.main.transform.rotation; // Get the camera's
    rotation
    weaponSlots = new List<WeaponData>(); // Clear the weapon slots
    foreach (GameObject weapon in
    player.GetComponent<PlayerGunInteraction>().weaponSlots) // Get a list every
    weapon currently equipped
    {
        Weapon weaponScript = weapon.GetComponent<Weapon>(); // Get the
        underlying Weapon script of the weapon
        weaponSlots.Add(new WeaponData(weaponScript.WeaponType,
        weaponScript.TotalAmmo, weaponScript.AmmoInClip, weaponScript.ClipCapacity));
        // Make new weapon save data and add it to the list
    }
    activeWeaponSlot =
    player.GetComponent<PlayerGunInteraction>().activeWeaponSlot; // Save the
    position of the currently equipped weapon
    merit = PlayerManager.Instance.merit; // Assign merit
```

```

        sanity = PlayerManager.Instance.sanity; // Assign sanity
        timeOnSave += Time.realtimeSinceStartup; // Add on the time that the
player has had the game open
        totalKills = PlayerManager.Instance.totalKills; // Get the total kills
the player has done
    }
}

[System.Serializable]
public class WeaponData
{
    public string name;
    public int totalAmmo;
    public int ammoInClip;
    public int clipCapacity;
    public WeaponData(string name, int totalAmmo, int ammoInClip, int
clipCapacity)
    { // Copies the important weapon data into a smaller format
        this.name = name;
        this.totalAmmo = totalAmmo;
        this.ammoInClip = ammoInClip;
        this.clipCapacity = clipCapacity;
    }
}

```

## WorldFlags.cs

```

[System.Serializable]
public class WorldFlags
{
    int seenTutorial = 0;
}

```

## SaveData.cs

```

using UnityEngine;

[System.Serializable]
public class SaveData
{
    public PlayerData playerData;
    public WorldFlags worldFlags;
    public LevelHighscores levelHighscores;
    public LevelBullets levelBullets;

    public SaveData()
    { // Initialises new save data with the default values
}

```

```

        playerData = new PlayerData();
        worldFlags = new WorldFlags();
        levelHighscores = new LevelHighscores();
        levelBullets = new LevelBullets();
    }
    public void UpdateSaveData(GameObject player)
    { // Update the player's save data
        playerData.UpdateData(player);
    }
}

```

## SaveSystem.cs

```

using UnityEngine;
using System.IO;
using System.Globalization;
using System.Text.RegularExpressions;
using System;
using Newtonsoft.Json;
using System.Collections.Generic;

public class CompareSaves : IComparer<string>
{
    public int Compare(string s1, string s2){
        return
int.Parse(Regex.Match(Path.GetFileNameWithoutExtension(s1).ToString(),
@"\d+$").ToString()) >
int.Parse(Regex.Match(Path.GetFileNameWithoutExtension(s2).ToString(),
@"\d+$").ToString()) ? 1 : -1; // if the number on the first string is larger
than the second, return 1, else return -1
    }
}

public static class SaveSystem
{
    public static void Save(SaveData data, string type, int saveSlot)
    {
        type = new CultureInfo("en-US", false).TextInfo.ToTitleCase(type); // Make sure the type is following the naming convention
        string path = Path.Combine(Application.persistentDataPath, "Saves",
"Slot" + saveSlot.ToString(), type); // Make the path to the folder
        string fileMatch = "Slot" + saveSlot.ToString() + type + "*.dat"; // Get the pattern for the important files in the folder
        string[] matches = Directory.GetFiles(path, fileMatch); // Get all files that match the pattern
        Array.Sort(matches, new CompareSaves()); // Sort the files based on the last number
        int fileNum = 1; // Default value if no files are found
        if (matches.Length > 0)

```

```
{  
    string lastFile = matches[^1]; // Get the latest file in the  
directory  
    lastFile = Path.GetFileNameWithoutExtension(lastFile).ToString();  
// Get only the name of the file  
    fileNum = Int32.Parse(Regex.Match(lastFile, @"\d+$").ToString()) +  
1; // Get the number of the file and increment it  
}  
path = Path.Combine(path, "Slot" + saveSlot.ToString() + type +  
fileNum.ToString() + ".dat"); // Make the name of the new file  
  
using (StreamWriter stream = new StreamWriter(path)) // Make the file  
on the system  
{  
    string json = JsonConvert.SerializeObject(data,  
Formatting.Indented); // Convert the save data into JSON  
    stream.WriteLine(json); // Fill with constructed save data  
}  
// Modify Lookup  
Dictionary<string, string> lookup = GetLookup(); // Read the lookup  
lookup[$"Slot{saveSlot}"] = path; // Edit this slot's new path  
List<string> keys = new List<string>(lookup.Keys); // Get all the keys  
List<string> values = new List<string>(lookup.Values); // Get all the  
values  
  
using (StreamWriter lookupWriter = new  
StreamWriter(Path.Combine(Application.persistentDataPath, "SaveLookup.txt")))  
{ // Write over SaveLookup.txt  
    for (int i = 0; i < keys.Count; i++) // For each key in the  
dictionary  
    {  
        lookupWriter.WriteLine(keys[i] + "," + values[i]); // Write  
each line with a comma as the separator  
    }  
}  
}  
public static SaveData LoadSave(string path)  
{  
    if (!File.Exists(path)){ // If the file doesn't exist  
        Debug.Log($"File at '{path}' does not exist.");  
    }  
    SaveData data; // Define the save data  
    using (StreamReader sr = new StreamReader(path))  
    { // Open the file for reading  
        data = JsonConvert.DeserializeObject<SaveData>(sr.ReadToEnd()); //  
Convert the file back into a SaveData class  
    }  
}
```

```
        return data; // Return the read data
    }
    public static List<SaveSlot> ListAllSlots()
    {
        Dictionary<string, string> lookupPairs = GetLookup(); // Get the slot
pairs
        if (lookupPairs == null) // If there are no saves made
        {
            return new List<SaveSlot>(); // Return an empty list
        }
        string path = Path.Combine(Application.persistentDataPath, "Saves");
// Path to the saves folder
        if (!Directory.Exists(path)) // If no saves have been made
        {
            Directory.CreateDirectory(path); // Create the folder
            return new List<SaveSlot>(); // Return an empty list
        }
        string[] directories = Directory.GetDirectories(path); // Get all the
folders within the Saves folder
        List<SaveSlot> slotList = new List<SaveSlot>(); // Create a new
foreach (string directory in directories)
{ // For every save slot
    string folder =
directory.Substring(path.Length).TrimStart(Path.DirectorySeparatorChar); // // Get the folder name
    string savePath = lookupPairs[folder]; // Get the filepath
    SaveSlot slot = new SaveSlot(savePath); // Create a new container
    slotList.Add(slot); // Add it to the list
}
        return slotList; // Return the SaveSlot list
    }
    public static Dictionary<string, string> GetLookup()
    {
        if (!File.Exists(Path.Combine(Application.persistentDataPath,
"SaveLookup.txt")))
        { // If the file doesn't exist
            File.Create(Path.Combine(Application.persistentDataPath,
"SaveLookup.txt")).Close(); // Create it and close it
        }

        string[] pathLookup =
File.ReadAllLines(Path.Combine(Application.persistentDataPath,
"SaveLookup.txt")); // Read every line and store as an array
        Dictionary<string, string> lookupPairs = new Dictionary<string,
string>(); // Make a new dictionary
        foreach (string lookup in pathLookup)
{ // For every line in the file
    string[] pairArray = lookup.Split(','); // Split it into pairs
```

```
        lookupPairs.Add(pairArray[0], pairArray[1]); // Add each pair into
the dictionary
    }
    return lookupPairs; // Return the dictionary
}
public static SaveData NewSave()
{
    if (!Directory.Exists(Path.Combine(Application.persistentDataPath,
"Saves"))) // If Saves doesn't exist
    {
        Directory.CreateDirectory(Path.Combine(Application.persistentDataP
ath, "Saves")); // Create it
    }
    string[] dirs =
Directory.GetDirectories(Path.Combine(Application.persistentDataPath,
"Saves")); // Get all Save slots
    Array.Sort(dirs, new CompareSaves()); // Sort the slots using the same
system
    int dirNum = 1;
    if (dirs.Length > 0)
    {
        string lastDir = dirs[^1]; // Get the last directory
        dirNum = Int32.Parse(Regex.Match(lastDir, @"\d+$").ToString()) +
1; // Get the number of the last slot
    }
    // Create the folders for the save slot
    string folderName = "Slot" + dirNum.ToString(); // Create the folder
name
    if (!Directory.Exists(Path.Combine(Application.persistentDataPath,
"Saves", folderName))) // If the folder doesn't exist
    {
        Directory.CreateDirectory(Path.Combine(Application.persistentDataP
ath, "Saves", folderName)); // Create the folder
        Directory.CreateDirectory(Path.Combine(Application.persistentDataP
ath, "Saves", folderName, "Manual")); // Create the Manual folder inside
        Directory.CreateDirectory(Path.Combine(Application.persistentDataP
ath, "Saves", folderName, "Auto")); // Create the Auto folder inside
    }
    else
    {
        throw new Exception($"The save folder {dirNum} was unable to be
created.");
    }
    // Create the save file
    SaveData data = new SaveData(); // Create new save data
    string path = Path.Combine(Application.persistentDataPath, "Saves",
folderName, "Manual", "Slot" + dirNum.ToString() + "Manual1.dat"); // Get the
path to the file
```

```

        using (StreamWriter saveWriter = new StreamWriter(path)) // Make the
file on the system
{
    string json = JsonConvert.SerializeObject(data,
Formatting.Indented); // Convert the save data into JSON
    saveWriter.Write(json); // Fill with constructed save data
}
// Modify the lookup
if (!File.Exists(Path.Combine(Application.persistentDataPath,
"SaveLookup.txt"))) // If the lookup doesn't exist
{
    File.Create(Path.Combine(Application.persistentDataPath,
"SaveLookup.txt")).Close(); // Make it and then close it
}
File.AppendAllText(Path.Combine(Application.persistentDataPath,
"SaveLookup.txt"), $"{folderName},{path}\n"); // Append the necessary pair
return data; // Return the save data
}
}

public class SaveSlot
{
    public string dataPath; // Exists so that SaveData only has 1 instance
open at any time for memory
    public string number;
    public string sceneName;
    public float time;
    public SaveSlot(string dataPath)
    {
        this.dataPath = dataPath; // Stores the path to the save on the disk
        number = Regex.Match(Path.GetFileName(dataPath),
@"(?=<Slot)\d+").ToString(); // Gets the ending number from every slot
        SaveData data = SaveSystem.LoadSave(dataPath); // Opens the save file
to read values
        sceneName = data.playerData.currentScene; // Gets the scene the save
was in
        time = data.playerData.timeOnSave; // Gets the amount of time the
player has spent in the save
    }
}

```

## SerializableStructs.cs

```

using UnityEngine;

[System.Serializable]
public struct SerVector3
{

```

```

public float x;
public float y;
public float z;

public SerVector3(float x, float y, float z)
{
    this.x = x;
    this.y = y;
    this.z = z;
}
public static implicit operator Vector3(SerVector3 v)
{ // This is used for casting from SerVector to Vector3
    return new Vector3(v.x, v.y, v.z);
}
public static implicit operator SerVector3(Vector3 v)
{ // This is used for casting from Vector3 to SerVector
    return new SerVector3(v.x, v.y, v.z);
}
}

[System.Serializable]
public struct SerQuaternion
{
    public float x;
    public float y;
    public float z;
    public float w;
    public SerQuaternion(float x, float y, float z, float w)
    {
        this.x = x;
        this.y = y;
        this.z = z;
        this.w = w;
    }
    public static implicit operator Quaternion(SerQuaternion q)
    { // This is used for casting from SerQuaternion to Quaternion
        return new Quaternion(q.x, q.y, q.z, q.w);
    }
    public static implicit operator SerQuaternion(Quaternion q)
    { // This is used for casting from Quaternion to SerQuaternion
        return new SerQuaternion(q.x, q.y, q.z, q.w);
    }
}

```

## GUISlots.cs

```

using System.Collections.Generic;
using UnityEngine;

```

```

using TMPro;
using System;

public class GUISlots : MonoBehaviour
{
    [SerializeField] GameObject slotPrefab;

    private List<SaveSlot> slots;
    private void Start()
    {
        DisplaySaves(); // Display the saves when the game starts
    }
    public void DisplaySaves()
    {
        for (int i = gameObject.transform.childCount - 1; i >= 0; i--)
        { // Destroy all existing visual slots, going backwards to avoid
errors
            Destroy(gameObject.transform.GetChild(i).gameObject);
        }
        slots = SaveSystem.ListAllSlots(); // Get all the slots
        foreach (SaveSlot slot in slots)
        { // For every slot
            GameObject slotInList = Instantiate(slotPrefab,
gameObject.transform); // Make a new object
            var image = slotInList.transform.GetChild(0); // Get the image
            slotInList.transform.GetChild(1).GetComponent<SelectSlot>().dataPa
th = slot.dataPath; // Set the datapath
            TMP_Text number =
slotInList.transform.GetChild(2).gameObject.GetComponent<TMP_Text>(); // Get
the number text element
            TMP_Text time =
slotInList.transform.GetChild(3).gameObject.GetComponent<TMP_Text>(); // Get
the time text element
            TMP_Text scene =
slotInList.transform.GetChild(4).gameObject.GetComponent<TMP_Text>(); // Get
the scene text element
            var saves = slotInList.transform.GetChild(5); // Get the saves
button
            number.text = slot.number; // Set the slot number
            time.text = TimeSpan.FromSeconds(slot.time).ToString(); // Set the
time in save
            scene.text = slot.sceneName; // set the scene name
        }
    }
}

```

## LevelLoading.cs

```
using System.Collections;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

public class LevelLoading : MonoBehaviour
{
    public Slider progressBar;
    public Button button;
    private AsyncOperation op;

    public void Load(string sceneName)
    {
        button.gameObject.SetActive(false); // Make sure that the button is disabled
        StartCoroutine(DisplayProgress(sceneName)); // Start loading
    }

    public void ShowButton()
    {
        if (button == null){ // If the button is not assigned
            button = transform.GetChild(2).GetComponent<Button>(); // Assign the button
        }
        button.gameObject.SetActive(true); // Enable the button
    }

    public void FinishLoad()
    {
        op.allowSceneActivation = true; // Switch to the new scene
    }

    IEnumerator DisplayProgress(string sceneName)
    {
        AsyncOperation load = SceneManager.LoadSceneAsync(sceneName); // Begin loading the scene in the background
        op = load; // Store so that the button can use later
        load.allowSceneActivation = false; // Make sure the level does not automatically replace the one
        Debug.Log($"Started scene load of scene '{sceneName}'");
        while (progressBar.value < 1) // While the bar has not finished - makes sure that the player sees a filled in bar
        {
            float progress = Mathf.Clamp01(load.progress / 0.9f); // Get the current progress

            progressBar.value = progress; // Assign the progress to the bar
            yield return new WaitForEndOfFrame(); // Wait for the end of frame
            if (progressBar == null){ // If the progress bar has been dereferenced

```

```

        progressBar = transform.GetChild(1).GetComponent<Slider>(); // Get the slider object
    }
}
ShowButton(); // Show the button
}
}

```

## NewGame.cs

```

using UnityEngine;
using UnityEngine.UI;

public class NewGame : MonoBehaviour
{
    private void Start()
    {
        Button button = gameObject.GetComponent<Button>(); // Get the button
        this is attached on
        button.onClick.AddListener(NewGameButton); // Listen for the player
        clicking it
    }
    public void NewGameButton()
    { // Make a new save
        PlayerManager.Instance.saveData = SaveSystem.NewSave();
    }
}

```

## SelectSlot.cs

```

using UnityEngine;

public class SelectSlot : MonoBehaviour
{
    public string dataPath;
    public void SelectThisSlot()
    {
        GameObject obj = GameObject.Find("Load"); // Find the load button
        SlotSelection button = obj.GetComponent<SlotSelection>(); // Get the
        selection script on it
        button.selected = gameObject; // Set this object as the most recently
        selected
    }
}

```

## SlotSelection.cs

```
using System;
```

```

using UnityEngine;
using TMPro;

public class SlotSelection : MonoBehaviour
{
    public GameObject selected;

    public void LoadSelected()
    {
        if (selected == null)
        { // If nothing has been selected, don't continue
            return;
        }
        SaveData data =
SaveSystem.LoadSave(selected.GetComponent<SelectSlot>().dataPath); // Get the
save data
        int slot =
int.Parse(selected.transform.parent.GetChild(2).gameObject.GetComponent<TMP_Te
xt>().text); // Get the slot number
        PlayerManager.Instance.SetSlotandData(slot, data); // Send the slot
number and data to the game manager
        PlayerManager.Instance.LoadSlot(selected.transform.parent.GetChild(4).g
ameObject.GetComponent<TMP_Text>().text); // Load the game slot
    }
}

```

## GlobalConstants.cs

```

public static class GlobalConstants
{
    public static float maxSlope = 60f;
}

```

## Interactable.cs

```

using UnityEngine;

public abstract class Interactable : MonoBehaviour
{
    void Start()
    {
        gameObject.tag = "Interact"; // Mark the current object as an
interactable
    }
    abstract public void Interact(); // Define the required Interact method
}

```